

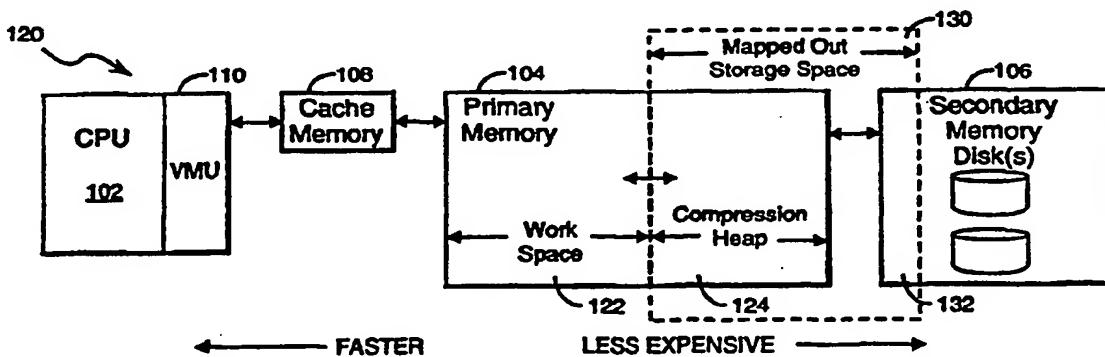
PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F	A2	(11) International Publication Number: WO 95/18997 (43) International Publication Date: 13 July 1995 (13.07.95)
<p>(21) International Application Number: PCT/US94/14987</p> <p>(22) International Filing Date: 29 December 1994 (29.12.94)</p> <p>(30) Priority Data: 08/177,622 30 December 1993 (30.12.93) US</p> <p>(71) Applicant: CONNECTIX CORPORATION [US/US]; 2655 Campus Drive, San Mateo, CA 94403 (US).</p> <p>(72) Inventors: GARNER, Jonathan, F.; 13503 Skyline Boulevard, Woodside, CA 94062 (US). BROWN, Jorg, A.; 1516 Wicket Court, Concord, CA 94518 (US). WALTERS, Chad, Perry; 550 Middlefield Avenue, Palo Alto, CA 94301 (US).</p> <p>(74) Agents: WILLIAMS, Gary, S. et al.; Flehr, Hohbach, Test, Albritton & Herbert, 4 Embarcadero Center, Suite 3400, San Francisco, CA 94111-4187 (US).</p>		(81) Designated States: AM, AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, JP, KE, KG, KP, KR, KZ, LK, LR, LT, LU, LV, MD, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SI, SK, TJ, TT, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG), ARIPO patent (KE, MW, SD, SZ).
<p>Published <i>Without international search report and to be republished upon receipt of that report.</i> </p>		

(54) Title: VIRTUAL MEMORY MANAGEMENT SYSTEM AND METHOD USING DATA COMPRESSION



(57) Abstract

A virtual memory system and method enable a computer system to use a virtual memory address space larger than the size of physical primary memory while swapping few, if any, pages out to secondary memory. Primary memory is divided into a work space, used for storing uncompressed pages in current use, and a "Compression Heap". A MappedOut storage space, which includes the Compression Heap and a portion of secondary memory, is used to store all pages swapped out of the work space. A virtual memory manager dynamically determines the number of pages of primary memory which need to be included in the work space, and moves pages of primary memory into the work space from the Compression Heap as needed. Pages are selected to be swapped out of the work space to the MappedOut storage space on the basis of memory usage data. A data compressor compresses at least some virtual memory pages prior to their storage in the MappedOut storage space, and a data decompressor decompresses the compressed virtual memory pages when a fault occurs on those pages. The virtual memory manager includes logic for preferentially compressing and storing swapped out virtual memory pages in the Compression Heap so long as the MappedOut storage page includes a sufficient number of pages of primary memory to store those virtual memory pages, and for storing ones of the swapped out virtual memory pages in secondary memory when insufficient space is available in the Compression Heap.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgyzstan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo	KR	Republic of Korea	SE	Sweden
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LU	Luxembourg	TD	Chad
CS	Czechoslovakia	LV	Latvia	TG	Togo
CZ	Czech Republic	MC	Monaco	TJ	Tajikistan
DE	Germany	MD	Republic of Moldova	TT	Trinidad and Tobago
DK	Denmark	MG	Madagascar	UA	Ukraine
ES	Spain	ML	Mali	US	United States of America
FI	Finland	MN	Mongolia	UZ	Uzbekistan
FR	France			VN	Viet Nam
GA	Gabon				

VIRTUAL MEMORY MANAGEMENT SYSTEM AND METHOD USING DATA COMPRESSION

A portion of the disclosure of this patent document contains materials to which a claim of copyright protection is made. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, but reserves all other rights whatsoever.

5

- The present invention relates generally to virtual memory management systems and methods, and particularly to a system and method for improving the performance of a virtual memory management system by utilizing part of a computer system's high speed random access memory (RAM) to store "swapped out" pages in compressed form.

10

BACKGROUND OF THE INVENTION

- 15 In most computers having virtual memory management, the available address space for the computer's operating system and application processes exceeds the amount of physical RAM available. As shown in Figure 1, a computer system 100 will typically have a central processing unit (CPU) 102, such as a Motorola 68040 or Intel 80486, a primary memory 104 (typically random access memory (RAM)), a secondary memory 106 (typically one or more magnetic or optical disks), and a cache memory 108 for storing a copy of the memory locations most recently addressed by the CPU 102. The cache memory 108, or a portion of the cache memory, is often fabricated as part of the same integrated circuit as the CPU 102.
- 20

- 2 -

As indicated in Figure 1, the CPU 102 is typically the fastest of the components in the computer system, followed by the cache memory 108, the primary memory 104 and finally the secondary memory 106. Also shown in Figure 1 is a virtual memory unit (VMU) 110, which is often fabricated as part of the same integrated circuit as the CPU 102.

Conventional Virtual Memory Management

- 10 The following is a brief explanation, for those readers not skilled in the art, of how conventional virtual memory management works. The virtual memory unit 110 enables programs executed by the CPU 102 to utilize an address space that is larger than the actual storage capacity of the primary memory 104. For instance, a computer with 8 MBytes (MegaBytes) of RAM might utilize a virtual address space of 16 MBytes, or even more. The VMU 110 uses a "page table" (discussed in more detail below) to store one "page table entry" for each virtual memory page in the computer's virtual memory address space. One version of a page table is shown in Figure 4 (although that page table contains features particular to the present invention).
- 15
- 20 Each page table entry includes an address value and status flags indicating whether the corresponding virtual memory page is stored in primary or secondary memory. Thus, the status flags in a page table entry indicate whether the address value in that entry corresponds to a primary memory location or a secondary memory location. For a computer system using pages that are 4 KBytes (KiloBytes) in size, a 16 MBytes virtual address space would occupy 4096 pages, and thus the page table would contain 4096 page table entries.
- 25
- 30 Since the CPU uses virtual addresses for its internal address computations, the virtual memory unit 110 translates each CPU specified virtual memory address values into a primary memory address value when the status flags in the

- 3 -

corresponding page table entry indicates that the CPU specified address is currently stored in the primary memory 104. The translated address signal generated by the virtual memory unit 110 is then used to address the primary memory 104. When the CPU specified virtual memory address value is not 5 currently stored in the primary memory 104, the page table entry accessed by the virtual memory unit will have a status flag indicating that fact, which causes the virtual memory unit to generate a "fault signal". The fault signal indicates the virtual page in which the fault occurred and also indicates, either directly or indirectly, the values stored in the address and flag fields of the corresponding 10 page table entry.

Whenever a virtual memory fault occurs, a software procedure, typically called a virtual memory manager, locates a page of primary memory that can be mapped into the virtual memory page in which the fault occurs. If the virtual page 15 upon which the fault occurred was previously swapped out to secondary memory, the contents of the virtual memory page are copied into the page of primary memory. The virtual memory manager then updates the page table entry for the virtual page to indicate the physical primary memory page in which the virtual memory page is now located, and then signals the virtual memory unit 110 to 20 resume operation. Upon resuming operation, the virtual memory unit 110 again accesses the page table entry for the CPU specified virtual memory address, and this time successfully translates the virtual memory address into a primary memory address.

25 While there are many other aspects of virtual memory management not discussed here, the salient point here is that in standard virtual memory management subsystems, at any one point in time a virtual memory page is either in primary memory or secondary memory, or possibly in transition from one to the other.

- 4 -

Basis for Present Invention

- During the period from 1983 through 1993, CPU operating speeds have increased much more dramatically than disk memory and other secondary memory 5 operating speeds. As a result, the "penalty" (on a computer system's operating speed) associated with the use of virtual memory has actually been getting worse, in that using a fast CPU causes more CPU cycles to be wasted when waiting for a page of virtual memory to be swapped in from secondary memory to primary memory.
- 10 Consider for a moment the relative speeds of (A) swapping a page out to disk (i.e., the prior art) and (B) compressing a page of 4096 bytes and storing it in a RAM heap for compressed pages (i.e., the present invention). Each disk access, almost regardless of the page size, takes an average of 9 to 20 15 milliseconds, depending on the disk's operating speed. We will assume the use of a moderately fast disk with a 10 millisecond access time. Tests of the data compression and decompression procedures used in the preferred embodiment indicate that a page of 2048 16-bit (2-byte) words (i.e., 4 KBytes) is compressed, on average, in 8 milliseconds, and is decompressed, on average, in about 2.3 20 milliseconds using a 25 MHz Motorola 68040. When using a 50 MHz Motorola 68030, the average compression and decompression execution times for a page are 12 milliseconds and 3.5 milliseconds, respectively.
- As indicated above, data decompression is much faster than data compression. 25 Thus, decompressing a compressed page takes much less time than recovering a swapped out page from disk (typically less than 25% as much time).
- Another basis for the present invention is the observation that the contents of primary memory (i.e., RAM) in most computer systems is much more 30 compressible than the average disk file. In particular, the inventors of the present invention have observed that the contents of the swappable pages in RAM are

- 5 -

typically compressible by a factor of three using standard Lempel-Zif data compression techniques and garbage collection of deallocated memory address ranges, while the contents of disk files are typically compressible by only a factor of 2 to 2.2 using standard Lempel-Zif data compression techniques. Given an

- 5 average compression ratio of 3:1, consider the following: if half (e.g., 4 MBytes) of a computer's 8-MByte RAM is used for normal memory access, and the second half of RAM (4 MBytes) is used to store compressed pages, then the total data content of RAM will be (4 MBytes + 12 MBytes = 16 MBytes), on average, twice the RAM's normal storage capacity.

10

Thus, using the present invention, it is possible to have a virtual memory address space that is twice the size of the computer's primary memory without having to swap any pages out to disk.

- 15 Of course, the contents of RAM are not always of average compressibility, and thus it may be necessary at times to swap some pages out to disk. However, as noted above, decompressing a page generally takes less than 25% as much time as a disk access, and thus even if a compressed page is swapped out to disk, the use of data compression slows the recovery of that particular page by
20 only about 25%. Furthermore, in normal usage less than 20% of pages removed from the "work space" will be swapped out to disk, and thus, on average, the time for recovering from page faults is substantially reduced. (Actually, in most cases, no pages will be swapped out to disk, with 20% representing the normal upper limit.) As shown by the following computation:

25

$$\begin{aligned} & 0.80 \text{ (percentage of pages)} \times 0.25 \text{ (relative speed of access)} \\ & + 0.20 \times 1.25 = 0.20 + 0.25 = 0.45 \end{aligned}$$

- the average page fault response time is reduced by about 50% using the present
30 invention with a CPU of only very average capability, while using less disk space than standard virtual memory subsystems. When using a CPU at least as

- 6 -

powerful as a 40 MHz 68040 or a 66 MHz 80486DX-2 microprocessor, the average page fault response time is reduced by 60% or more, with correspondingly better performance for even faster processors.

- 5 Another motivation for the present invention is that hard disk use is very power consuming for portable computers. Normally, using virtual memory on a portable computer would mean that the hard disk would be in almost constant use, thereby draining the computer's battery. Use of the present invention allows the use of virtual memory while eliminating or substantially reducing the power usage
10 associated with swapping pages to and from hard disk.

The present invention is generally applicable to any computer system having a virtual memory unit and a CPU at least as powerful as a Motorola 68030 or an Intel 80386 running at 25 Megahertz. In such computer systems, use of the
15 present invention will reduce the latencies associated with the use of virtual memory and thus will improve overall system performance. In addition, the present invention will enable many computer users to avoid the purchase of additional random access memory since the "penalty" associated with using virtual memory instead of physical random access memory will be reduced.
20

SUMMARY OF THE INVENTION

- In summary, the present invention is a virtual memory system and method for
25 enabling a computer system to use a virtual memory address space that is larger than the size of physical primary memory while swapping few, if any, pages out to secondary memory. The computer's primary memory is divided into a work space, used for storing uncompressed pages that are in current use by the computer system's operating system and applications, and a "Compression
30 Heap". A MappedOut storage space, which includes the Compression Heap

- 7 -

and a portion of the computer's secondary memory, is used to store all pages swapped out of the work space.

A virtual memory manager dynamically determines the number of pages of primary memory which need to be included in the work space, and moves pages of primary memory into the work space from the compression heap as needed. Virtual pages are selected to be swapped out of the work space to MappedOut storage on the basis of memory usage data. The virtual memory manager includes logic for preferentially compressing and storing swapped out virtual memory pages in the Compression Heap (in primary memory) so long as the MappedOut storage space includes a sufficient number of pages of primary memory to store those virtual memory pages, and for storing ones of the swapped out virtual memory pages in secondary memory when insufficient space is available in the Compression Heap. Thus, the size of the work space is variable and the proportion of swapped pages stored in primary memory and in secondary memory are also variable.

A page table stores a page table entry corresponding to each virtual memory page. Each page table entry specifies a tag value, corresponding to a location in the primary or secondary memory, and a plurality of status flags for indicating the status of the corresponding virtual memory page. The status flags includes a MappedIn flag having a True/False value that indicates when the tag value corresponds to a RAM page (in the active work space) in which the corresponding virtual memory page is stored.

A virtual memory unit (VMU) translates CPU specified virtual memory address values into primary physical address values when the MappedIn flag in the page table entries corresponding to the CPU specified virtual memory address values are True. The VMU generates fault signals when the MappedIn flag in the page table entries corresponding to the CPU specified virtual memory address values are False.

- 8 -

- The virtual memory manager responds to each fault signal from the VMU by updating the page table entry corresponding to the CPU specified virtual memory address value which caused the fault signal, so that the page table entry specifies a primary memory page and a MappedIn flag having a True value. The virtual
- 5 memory manager includes a work space manager that defines a work space and a MappedOut storage space for storing virtual memory pages. The work space is located in a first portion of the primary memory. The MappedOut storage space is located in a second portion of the primary memory pages and in a portion of the secondary memory.
- 10
- A memory usage monitor stores memory usage data. The virtual memory manager includes swap out selection logic that selects, on the basis of the memory usage data, virtual memory pages to be swapped out from the work space to the MappedOut storage space. A MappedOut storage manager receives
- 15
- from the virtual memory manager the virtual memory pages selected to be swapped out, stores the received virtual memory pages in the MappedOut storage space, and adds the primary memory pages in which the received virtual memory pages were stored to the MappedOut storage space.
- 20
- The MappedOut storage manager includes a data compressor that compresses at least some of the received virtual memory pages prior to their storage in the MappedOut storage space, and a data decompressor that decompresses the compressed virtual memory pages when the virtual memory manager responds to fault signals caused by VMU faults on the compressed virtual memory pages.
- 25

BRIEF DESCRIPTION OF THE DRAWINGS

- Additional objects and features of the invention will be more readily apparent
- 30
- from the following detailed description and appended claims when taken in conjunction with the drawings, in which:

- 9 -

Figure 1 is a block diagram of the memory hierarchy in a conventional computer system.

5 Figure 2 is a block diagram showing the memory hierarchy in a computer system using the present invention.

Figure 3 schematically represents a typical virtual memory address map for a computer system using the present invention.

10 Figure 4 is a block diagram of a page table (also herein called a page map) for a computer system using the present invention.

Figure 5 is a diagrammatic representation of one page table entry in the page table of Figure 4.

15 Figure 6 is a diagrammatic representation of the virtual memory manager used in a preferred embodiment of the present invention.

20 Figure 7 is a diagrammatic representation of a memory usage table that stores data indicating which virtual memory pages in a defined virtual memory address space were least recently used.

25 Figure 8 is a diagrammatic representation of a table used in a preferred embodiment of the present invention to store data representing a list of "best" pages to swap out from the computer's work space if it becomes necessary to swap pages out of the work space.

Figure 9 is a diagrammatic representation of the Compression Manager used in a preferred embodiment of the present invention.

30

- 10 -

Figure 10 is a diagrammatic representation of a "Sector Table" used in a preferred embodiment of the present invention to store data indicating which sectors of a secondary memory devices have been used to store swapped out virtual memory pages.

5

Figure 11 is a diagrammatic representation of a "Main Map" table used in a preferred embodiment of the present invention to store data representing all virtual memory pages that have been swapped out of the computer's work space.

10 Figure 12 is a diagrammatic representation of a "Map Chain" table used in a preferred embodiment of the present invention to store data indicating the order in which certain virtual memory pages were swapped out of the computer's work space.

15 Figure 13 is a diagrammatic representation of the "Compression Heap" used in a preferred embodiment of the present invention to store virtual memory pages swapped out of the computer's work space.

20 Figure 14 is a diagrammatic representation of the "Last Use" table used in a preferred embodiment of the present invention during the process of performing a data compression procedure on a virtual memory page.

25 Figure 15 is a diagrammatic representation of the "Index Table" table used in a preferred embodiment of the present invention during the process of performing a data compression procedure on a virtual memory page.

Figure 16 is a flow chart representation of the data compression procedure used in a preferred embodiment of the present invention.

30

- 11 -

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Tables 1-16 at the end of the specification contain pseudocode representations of software procedures relevant to the present invention. The pseudocode used in these appendices is, essentially, a computer language using universal computer language conventions. While the pseudocode employed here has been invented solely for the purposes of this description, it is designed to be easily understandable by any computer programmer skilled in the art. The computer programs in the preferred embodiment are written in the assembly language used for Motorola 680x0 microprocessors.

Referring to Figure 2, a computer system 120 in accordance with the present invention includes a central processing unit (CPU) 102, such as a Motorola 68040 or Intel 80486, a primary memory 104 (typically random access memory (RAM)), a secondary memory 106 (typically one or more magnetic or optical disks), and a cache memory 108 for storing a copy of the memory locations most recently addressed by the CPU 102.

Solely for the purposes of describing a preferred embodiment, it will be assumed that the CPU is a Motorola 68040 running at 33 Megahertz, that the primary memory 104 contains 8 MBytes of RAM, and that the secondary memory 106 is a fast hard disk having a capacity of over 100 MBytes. Furthermore, it will be assumed that the computer system 120 used in the preferred embodiment is a Macintosh computer, produced by Apple Computer Corporation, running the Macintosh 7.1 operating system. It should be understood that this system configuration is not part of the present invention, but rather is simply one example of a context in which the present invention can be used.

As shown in Figure 2, the primary memory 104 is divided into two portions: a Work Space 122 and a Compression Heap 124. A MappedOut storage space 130, which includes the Compression Heap 124 and a portion 132 of secondary

- 12 -

memory, is used to store all pages swapped out of the work space. The portions of primary memory allocated to the Work Space 122 and Compression Heap are variable, as will be explained below, and thus these portions of primary memory may not have fixed sizes.

5

In a preferred embodiment using a virtual address space that is precisely twice the size of the physical primary memory, the portion 132 of secondary memory 106 that is included in the MappedOut storage space is equal in size to the physical primary memory. Thus, in the preferred embodiment, for a computer 10 system with 8 MBytes of RAM, the virtual memory address space will be 16 MBytes and the portion 132 of secondary memory included in the MappedOut storage space will be 8 MBytes.

15

Virtual Memory Address Map and VM Page Table

Figure 3 shows a map 140 of the virtual address space in a computer using the present invention. The lowest addresses (i.e., near address zero) in the virtual address space are typically occupied by items that must always be present in 20 primary memory:

- the operating system's kernel 142,
 - interrupt vector tables 144,
 - the Page Table 146 (herein also called the Page Map) used by the virtual memory unit 110 to map virtual memory addresses into physical memory addresses,
 - a disk driver 148, needed for sending pages of virtual memory to and from disk, and
 - a virtual memory manager 150, for responding to virtual memory unit fault signals.
- 25 30 To that "standard" set of unswappable items, the present invention adds a virtual memory Compression Manager 152, which can be thought of as part of the virtual

- 13 -

memory manager. In the preferred embodiment, however, the VM Compression Manager 152 is a layer of software that is called by the VM Manager 150.

As shown in Figure 3, a first portion of the virtual memory address space extends
5 to an address that is twice as large as the physical RAM. Note that the variable "PhyMem" is equal to the size of the physical RAM, in units of bytes and that "2 x PhyMem" therefore means a value equal to twice the size of the physical RAM. This first portion of the virtual memory address space is used in the same way that conventional virtual memory is used: to store programs and data that
10 can be directly addressed by programs executed by the CPU.

The Work Space 122 comprises the unswappable items discussed above, plus all other pages of the "regular" virtual memory address space which are uncompressed and resident in primary memory.

15 A second portion of the virtual memory address space, located between virtual memory addresss "40 000 000 H - 2 x PhyMem" and 40 000 000 H (1 Gigabyte), is used by the VM Compression Manager 152 to store (A) a set of Compression Manager Tables 160 that are used to keep track of virtual memory pages stored
20 in the MappedOut storage space 130, and (B) the Compression Heap 124. Thus, the Compression Heap 124 is assigned to a portion of the virtual address space that is not directly addressable by ordinary programs executed by the CPU.

One page of virtual address space, just below the Compression Manager Tables
25 160 (but which could be located in any other page of unusued virtual memory address space), is herein called the Transient Page 158. Whenever a physical RAM page is about to be mapped into a specified virtual memory address, in response to a page fault, the page is first mapped into the address of the Transient Page. That address is herein labeled the TransientPageAdr. The page
30 is then usually filled with zeros, or with the previous contents of the virtual page,

- 14 -

and then the page is mapped into the specified virtual memory address for satisfying the page fault.

- The amount of virtual memory address space assigned to the Compression Manager Tables 160 depends on the maximum number of virtual memory pages that can be in used, excluding virtual memory pages used by the Compression Manager Tables 160 and the Compression Heap 124. In the preferred embodiment it is assumed that the virtual memory address space useable by operating system and application programs (i.e., all programs excluding the VM Manager and VM Compression Manager) will not exceed twice the physical primary memory.

- As will be described in more detail below, the Compression Manager Tables 160 include a Sector Table 162, Dictionary Tables 164 (including a LastUse table 166 and Index Table 168) having a fixed size of 136 KBytes, a Compression Buffer 170 of 6 KBytes, a Main Map table 172 and a Map Chain table 174. For a computer having 8 MBytes of physical RAM and 4-KByte pages, the available virtual address space is 4K pages, the Sector Table occupies 16 KBytes, the Main Map table 172 occupies 24 KBytes and the Map Chain table 174 occupies 16 KBytes, and thus the Compression Manager Tables 160 occupy a total of 192 KBytes (48 pages). The 48 pages in which the Compression Manager Tables 160 are stored, plus the first two pages assigned to the lowest addresses of the Compression Heap 124, are unswappable pages that must be kept in primary memory at all times.

25

- Figure 4 shows the virtual memory Page Table 146 used by the virtual memory unit VMU 110 (see Figure 2) to translate virtual addresses into physical memory addresses. The Page Table 146 is stored in RAM and includes one Page Table Entry (PTE) 180 for each defined virtual memory page. The Page Table is indexed by the most significant bits of the virtual memory address, as determined by the page size. For a computer using 32-bit addresses and 4-KByte pages,

- 15 -

the Page Table will be indexed by the 20 most significant bits of each virtual memory address. Thus, when the CPU tries to access any specified virtual memory address, the VMU will use the 20 most significant bits of that address to access a PTE in the Page Table. That 20-bit value is called the page number.

5

In the preferred embodiment each Page Table Entry 180 occupies 4 bytes and includes 20-bit Tag Value 182 and 12 status flags, only six of which are discussed herein. The 20-bit Tag Value 182 is equal to the 20 most significant bits of a physical address when the corresponding virtual memory page is mapped into 10 a page of RAM in the Work Space. Otherwise, the 20-bit Tag Value 182 is either a Tag Value used by the VM Compression Manager 152 to keep track of where the virtual memory page is located in the MappedOut storage space 130, or is a void value having no significance (e.g., if the virtual memory page is currently unassigned to any program or task).

15

Figure 5 represents a single Page Table Entry 180. The six status flags relevant to the present invention are as follows. The MappedIn (MI) flag 184 is set equal to 1 when a virtual memory page is mapped into the Work Space (the uncompressed part of primary memory) and set equal to 0 otherwise. The Tag 20 Value 182 is a physical RAM address only when MI=1.

The Modified (MD) flag 186 of a page is automatically set by the VMU 110 hardware whenever a write operation is performed to any location in that page. The Used (U) flag 188 of a page is automatically set by the VMU 110 hardware 25 whenever a read or a write operation is performed on any location in that page. Thus, the VMU 110 stores two types of memory usage information in the Page Table.

The MappedOut (MO) flag 190 is used by the VM Manager 150 of the present 30 invention to denote pages that have been mapped out of the Work Space. Pages

- 16 -

whose PTE has MO=1 may be either in the Compression Heap or in secondary memory.

- 5 The NoSwap (NS) flag 192 is used by the VM Manager 150 to mark virtual memory pages that must be kept resident in primary memory. These pages are indicated generally in Figure 4.

10 The WorkSpaceUsed (WSU) flag 194 is a cumulative use flag used by the VM Manager to keep track of all virtual memory pages used during a particular period of time herein called a Epoch. In the preferred embodiment an Epoch is about three seconds long, excluding time spent swapping pages out of the Work Space.

VIRTUAL MEMORY MANAGER

- 15 Referring to Figure 6, the VM Manager 150 includes a number of procedures 200-216 that will be described below. The VM Manager 150 maintains two data structures 220-222 for storing memory usage information. The LRU table 220 stores a 16 bit value for each physical RAM page, indicating which pages were least recently used, and the Best128List 222 is a list of up to 128 virtual memory pages that are candidates for being swapped out of the Work Space. The VM Manager also stores a number of miscellaneous pointers and data values 224 used to denote that status of various tasks the VM Manager performs.
- 20 The following is a description of the VM Manager procedures used in a preferred embodiment of the present invention.

Receive Fault - VMM Procedure

- 30 Referring to Table 1, the Receive Fault procedure is the top level procedure for processing a received page fault. The Receive Fault procedure receives the

- 17 -

page number (Page#) of the virtual memory page on which the fault occurred, the page's tag value (tag) and the status flags from the page's Page Table Entry.

- The primary job of the Receive Fault procedure is to call the GetPage procedure
- 5 202, which gets a page of RAM to satisfy the fault and maps that page into the virtual page address for the Transient Page. If the faulting page has status flags MI=0 and MO=0, that virtual memory page is currently unused, and thus it does not need to be swapped in from the MappedOut storage space 130.
- 10 Normally, the RAM page used to satisfy the fault on a page with MI=0 and MO=0 must be initialized, typically by filling it with zeros. This is (A) to make the page highly compressible if the page is later swapped out, and (B) to make sure that data left in the RAM page is not transferred to another application (which is a concern mostly in multiprocessing computer systems). However, if the MD status
- 15 flag is equal to 1, the page is not initialized because another VM Manager procedure (the Read Pending procedure) sets a page's status flags MI, MD, MO = 0, 1, 0 only when that page is about to be filled with data from an external source, such as a disk or network.
- 20 When a page fault occurs on a virtual page that is in the MappedOut storage space, the status flags MI, MD, MO = 0, 0, 1. In this case, after the GetPage procedure installs a RAM page into the Transient Page address, the contents of the virtual page are retrieved and copied into this RAM page by calling the ReadCompPage procedure. The contents of the previously mapped out page
- 25 are then flushed from MappedOut storage by calling the ReleaseCompPage procedure.

- Finally, the physical RAM page previously mapped into the Transient Page address is mapped into the virtual memory page at which the page fault occurred
- 30 by calling the MapPage procedure 203.

- 18 -

- Every time a page fault occurs, the ReceiveFault procedure also checks to see if the WSU status flags in the Page Table need to be cleared. In particular, each time a three second timer called ElapsedEpochTimer expires the WSU flags are cleared and the timer is restarted. In addition, whenever a foreground context
- 5 change occurs the WSU flags are cleared. The use of the WSU flags will be discussed below when the BuildBest128List procedure is explained.

Get Page - VMM Procedure

- 10 Referring to Table 2, the GetPage procedure 202 basically repeats two steps until the Compression Manager 152 hands it a free physical RAM page that can be mapped into the virtual memory page upon which a fault has occurred. The first step is to call the Compression Manager procedure called GetCompPage(PageAdr,fail). If the Compression Manager 152 has more than
- 15 two pages of free space in its Compression Heap 124, it returns the currently virtual address (PageAdr) of a RAM page at the end of the Compression Heap 124, and a fail flag equal to False. If the Compression Manager has less than two pages of free space in the Compression Heap 124, it returns a fail flag equal to True.
- 20 If the call to GetCompPage results in the receipt of a RAM page from the Compression Manager, the VM Manager procedure GetPage (A) maps that RAM page into the page table entry for the Transient Page, and (B) un-maps the RAM page from the end of the Compression Heap.
- 25 If the call to GetCompPage results in a fail flag signal from the Compression Manager, the VM Manager procedure GetPage calls the SwapOutPage procedure 204, which swaps one page from the active Work Space to the MappedOut storage space, and then the GetPage procedure once again calls GetCompPage
- 30 to try to get a free RAM page to map into the virtual memory page upon which

- 19 -

a fault has occurred. GetPage repeatedly calls GetCompPage and SwapOutPage until a free RAM page is received from the Compression Manager.

- Each call to the SwapOutPage procedure swaps one of the least recently used
- 5 pages in the active Work Space to the MappedOut storage space. Thus, each call to SwapOutPage reduces the Work Space by one page. However, the Compression Manager attempts to compress each page it receives from the SwapOutPage procedure, and thereby generates "free" RAM pages to hand back to the VM Manager. For instance, if the Compression Manager receives two
- 10 swapped out pages, and compresses each by a ratio of 3:1, the two compressed pages will occupy just two-thirds of a page, and thus the Compression Manager will have generated one and one-third free pages of storage space.
- In one circumstance the SwapOutPage procedure does not swap a page out
- 15 to MappedOut storage, and instead hands a RAM page directly over to the GetPage procedure. This happens when the page selected by the SwapOutPage procedure to swap out is an uninitialized page, which means that while a RAM page is mapped into the selected virtual memory page, the virtual memory page has not been used. In this case the SwapOutPage procedure returns the virtual
- 20 address of this uninitialized page, and GetPage then maps that page into its new virtual address (i.e., TransientPageAdr).

MapPage - VMM Procedure

- 25 The MapPage procedure (A) maps the physical RAM page previously obtained by the GetPage procedure into the page table entry for the virtual memory page on which the fault occurred, and (B) updates the "#UncompressedPages" value maintained by the VM Manager to indicate the increased number of pages in the Work Space, and then returns control to the ReceiveFault procedure.

30

- 20 -

Swap Out Page - VMM Procedure

- Referring to Table 3, the SwapOutPage procedure 204 makes use of the Best128List 222. Each time the SwapOutPage procedure is called, a swap counter, #Swaps, is incremented and compared with the SwapThreshold value.
- 5 When the #Swaps is greater than the SwapThreshold, the AgingEvent procedure 206 is called and the swap counter #Swaps is reset to zero.
- As discussed below, the AgingEvent procedure 206 updates the Best128List, which is a list of pages that are candidates for being swapped out to the MappedOut storage space. The AgingEvent procedure also updates a flag called GrowWS. GrowWS is True when the VM Manager has determined that the Work Space needs more primary memory pages, and is False otherwise.
- 10
- 15 The SwapOutPage procedure contains a loop that executes repeatedly until a page is found that can be swapped out of the Work Space. In the first part of the loop a page to be swapped out is selected. This is accomplished by looking through the pages listed in the Best128List. If the GrowWS flag is True, the pages listed in the Best128List are inspected and the first page in that list which has a True MappedIn status flag and a False Used flag is selected. Thus, any page in the Best128List which is currently in the WorkSpace and which was not recently in use is eligible to be swapped out. If the GrowWS flag is False, the pages listed in the Best128List are inspected and the first page in that list which has a True MappedIn status flag, a False Used flag, but which does not have
- 20
- 25 MI,MD,MO=1,0,1, is selected. The logical address of the selected page is stored in the variable PG.
- The status flag value MI,MD,MO=1,0,1 is a special status flag combination for pages that were previously sent to the Compression Manager and which returned by the Compression Manager as being "incompressible". Pages are denoted as being incompressible when the Compression Manager is unable to compress
- 30

- 21 -

the page by more than a predefined percentage, such as 12.5 percent. Pages known to be incompressible are swapped out only when the GrowWS flag is True.

- 5 If the entire Best128List is inspected and no eligible page for swapping out is found, the SwapOutPage procedure calls the BuildBest128List procedure 208, which regenerates the Best128List based on the current page usage information. The search for a page eligible to be swapped out is then repeated through the regenerated Best128List. This search will be successful, because the 10 regenerated Best128List will include only pages whose Used flags are False, unless the only pages in the Best128List are pages previously marked as incompressible (which is very unlikely to ever be the case). Since the LRU table's usage data for incompressible pages is set to the maximum possible value (FFFF) when the SwapOutPage procedure first tries to swap out those pages, such 15 pages rarely make it back to the Best128List.

However, if the Best128List still doesn't have any pages eligible for being swapped out, the AgingEvent procedure is called, which updates the LRUtable upon which the Best128List is based and also rebuilds the Best128List. The 20 search through the Best128List for a page eligible to be swapped out is then repeated. This process of searching the Best128List and calling the AgingEvent procedure if no swappable pages are found is repeated until a swappable page is selected.

- 25 Once a swappable page has been selected, the aging data for the selected physical RAM page is set to FFFF, which ensures that this RAM page will not be selected again in the near future for swapping out again. In addition, if the selected page is an uninitialized page, indicated by MI,MD,MO=1,0,0, this page can be directly remapped to a new location in the virtual address space and does 30 not need to be sent to the compression manager. Thus, when an uninitialized page is selected, its current virtual address is stored in the PageAdr return

- 22 -

argument, the ElapsedEpochTimer is reset to the value it had when SwapOutPage was called, and the SwapOutPage procedure exits, returning control to the GetPage procedure.

- 5 The second part of the SwapOutPage control loop is to swap out the selected page by calling the WriteCompPage procedure using parameters PG, GrowWS, Tag, and Fail, where PG is the logical address of the page to be swapped out, and GrowWS is the flag indicating if the Work Space needs to increase in size. Tag is a tag value returned by the Compression Manager that must be stored
- 10 in the Page Table in order to be able to recover the swapped out page from the MappedOut storage space, and Fail is a flag that is set when (A) the page selected to be swapped out is determined to be incompressible and (B) the GrowWS flag is False.
- 15 The WriteCompPage procedure is allowed to return a Fail value of True only when GrowWS is False. If WriteCompPage returns a Fail value of False, the page has been stored in the MappedOut storage space and the SwapOutPage procedure marks the page as being in the MappedOut storage space by (A) storing the returned Tag value in the Page Table Entry for the swapped out
- 20 page, and by storing status flags MI,MD,MO=0,0,1 in the Page Table Entry for the swapped out page. In addition, when a page is swapped out to the MappedOut storage space, the SwapOutPage procedure decreases the #UncompressedPages count value of pages in the Work Space by one.
- 25 If WriteCompPage returns a Fail value of True, the SwapOutPage procedure marks the page as incompressible by storing status flags MD,MO=0,1 in the Page Table Entry for that page. Furthermore, the process of selecting a page to swap out and then attempting to swap out that page is repeated until a page is successfully swapped out to the MappedOut storage space.

- 23 -

The ElapsedEpochTime timer is reset at the end of the SwapOutPage procedure to the time value it had when the SwapOutPage was called, thereby removing time spent in the SwapOutPage procedure from the measurement of the elapsed time for each work space usage epoch.

5

Aging Event - VMM Procedure

The LRU table 220 contains one 16-bit value for each physical RAM memory page. The data in the LRU table 220 is generated by shifting the Used status bits (U) from the Page Table into the LRU table at a rate such that the contents of the LRU table are completely replaced each time the number of swapped out pages equals the number of pages in the Work Space. If the number of uncompressed pages (#UncompressedPages) in the Work Space is 2048, then the SwapOutPage procedure generates a SwapThreshold value of $2048/16 = 128$ swaps. When the number of calls to the SwapOutPage procedure exceeds this SwapThreshold, the SwapOutPage procedure calls the AgingEvent procedure 206.

Referring to Table 4, the AgingEvent procedure 206 makes one pass through the Page Table 146 to collect the Used status bits in all the virtual memory pages that are currently mapped into the Work Space. Thus, pages that are designated as being unswappable by a NS status flag equal to 1, pages in MappedOut Storage and uninitialized non-resident virtual memory pages are ignored. For each virtual memory page in the Work Space, the physical RAM page number is obtained from that page's Page Table Entry. That RAM Page number is then used as an index to access an entry in the LRUtable 220, and that entry in the LRUtable is updated by right shifting the Used status bit of the virtual memory page in the LRUtable entry. Thus, the least significant bit of the LRUtable entry is discarded and the most significant bit of the LRUtable entry is set equal to the Used status bit.

- 24 -

From the above it can be seen that the pages least recently used will have the smallest values stored in the LRUtable.

In addition, for each virtual memory page in the Work Space, (A) the Used status
5 bit from the Page Table Entry is logically ORed with the Work Space Used status
bit (WSU) and stored as the updated WSU status bit in the Page Table Entry,
(B) the Used status bit is cleared, and (C) a variable called #PagesUsed is
increased by "1" if that page's WSU status bit is equal to "1". The variable
10 #PagesUsed represents the number of swappable virtual memory pages that
have been accessed since the last time all the WSU status bits were reset.

As the data in the LRUtable is updated, the AgingEvent procedure also
regenerates the Best128List. The Best128List 220 is a table having 128 slots
for storing virtual page numbers, plus a count #BestEntries indicating the number
15 of entries in the Best128List, and an age value BestAge, indicating the value
in the LRUtable for the pages listed in the Best128List.

When the AgingEvent procedure begins, the #BestEntries variable is set to zero
and the BestAge is set to FFFF. Each time an LRUtable entry is updated, the
20 LRUtable value is compared with the BestAge value. If the LRUtable entry is
less than the BestAge, the Best128List is restarted by setting BestAge to the
LRUtable entry's value, storing the current virtual page number if the first slot
of the Best128List, and by setting the #BestEntries to 1. If the LRUtable entry
is equal to the BestAge, the current virtual page is added to the Best128List by
25 storing its virtual page number in the slot of the Best128List designated by the
#BestEntries value, and by then increasing the #BestEntries by 1.

Finally, after all the virtual memory pages have been processed and the
Best128List regenerated, the GrowWS flag is regenerated by setting this flag
30 to True if the value of #PagesUsed (computed by the AgingEvent procedure)

- 25 -

is greater than the value of #UncompressedPages (which is the number of swappable pages in the Work Space).

- The order in which the Page Table Entries in the Page Table are processed by the AgingEvent procedure affects the order in which virtual pages are represented in the Best128List. To prevent any bias for swapping out pages in any particular portion of the virtual address space, the AgingEvent procedure maintains a pointer, StartSweepPtr, that is shifted by one Page Table Entry each time the AgingEvent or BuildBest128List procedure is called. The StartSweepPtr designates the page table entry at which the sweep through the Page Table begins for updating the LRUtable and for regenerating the Best128List.

Build Best 128 List - VMM Procedure

- 15 Referring to Table 5, the BuildBest128List procedure 208 is virtually the same as the AgingEvent procedure 206, except that the BuildBest128List procedure does not update the aged memory usage data in the LRUtable and does not clear the Used status bits in the Page Table. In addition, if the Best128List generated by the BuildBest128List has zero entries (which can happen if all pages
20 in the Work Space have been used since the last time the Used status flags were cleared by the AgingEvent procedure), the BuildBest128List procedure calls the AgingEvent procedure to update the LRUtable, clear the Used status bits in the Page Table, and then rebuild the Best128List of potentially swappable pages.

25 Read Pending - VMM Procedure

Referring to Table 6, the Read Pending procedure is called each time a Read from an external source occurs. A Read operation occurs whenever a disk read occurs, or a read from another I/O port is performed. The Read Pending procedure provides special handling for pages of virtual memory in MappedOut Storage that are about to be overwritten by data from an I/O device. In particular,

- 26 -

- if the virtual memory page that is about to receive data from an I/O read has MI,MO=0,1, there is no point in restoring the page of data stored in MappedOut Storage, because that data is about to be overwritten. The Read Pending procedure (A) releases the previously stored virtual memory page from the
- 5 MappedOut store by calling the ReleaseCompPage procedure, and (B) stores status flags of MI,MD,MO=0,1,0 in the Page Table Entry for each of the pages that the read operation will entirely fill with data. This status flag setting is a special marker indicating that the virtual page need not be initialized when a new RAM page is mapped into this virtual memory page, because it will be filled with
- 10 data from an upcoming read operation.

The Read Pending procedure also clears the MD and MO bits for all virtual memory pages that are currently mapped into the Work Space that are about to receive data from an I/O read so as to indicate that these pages are mapped in, but uninitialized. Clearing the Modified status flag (MD) also helps to ensure that the contents of these pages does not get stored in the MappedOut storage space before the read operation is completed.

15 After these Page Table operations are performed, processing returns to the
20 operating system read procedure that normally handles read operations.

Deallocate Pages and Deallocate Memory Range - VMM Procedures

Virtual memory pages are typically deallocated when an application terminates,
25 or when an application completes a task and releases the memory used for that task. Like the Read Pending procedure described above, the VM Manager's DeallocatePages procedure performs special Page Table operations to ensure proper handling of page deallocations. In particular, referring to Table 7, when
30 pages in MappedOut Storage are deallocated, the DeallocatePages procedure releases the previously stored virtual memory page from the MappedOut store

- 27 -

by calling the ReleaseCompPage procedure, and (B) stores status flags of MD, MO, U and WSU=0,0,0,0 in the Page Table Entry for release page.

- The Page Table Entries for deallocated pages mapped into the Work Space are modified by setting the MD, MO, U and WSU status bits to zero. In addition, these deallocated pages are filled with zeros (or any other repeating word value) to make the page highly compressible. For instance, if the page is later reused, but only a small portion of the page is used before it is swapped out, the unused portions of the page will be very highly compressible because the page was pre-filled with compressible data (i.e., cleared) upon its prior deallocation. Finally, a value of 0000 H is stored in the LRUtable entry for the deallocated pages, thereby making the RAM pages in which these virtual memory pages are stored highly eligible for reuse.
- Address ranges not beginning and ending on page boundaries can be deallocated at any time by application and system programs for a wide variety of reasons. The DeallocateMem procedure fills any partial pages in the deallocated address range with compressible data (e.g., zeros), and calls the DeallocatePage procedure (discussed above) to handle the deallocation of any complete pages in the deallocated address range.

VMM Initialization Procedure

- Referring to Table 8, the VMM Initialization Procedure is executed during or at the end of the computer's boot sequence. This procedure extends the Page Table to include the virtual memory address space needed for the Work Space and for the Compression tables and Compression Heap. It also installs the VM Manager, VM Compression Manager, and the Compression Manager Tables, assigns an initial set of RAM pages to the Compression Heap, and assigns all remaining RAM pages to the next logical addresses to be assigned by the operating system.

- 28 -

In multiprocessing implementations, the remaining RAM pages would be added to the end of the Compression Heap, which acts as a RAM page free list.

- 5 The NS, U and WSU status flags in the Page Table are initialized to indicate the unswappable pages used by the VM Manager and VM Compression Manager and to clear the Used and WorkSpaceUsed status flags in all Page Table Entries.

- 10 The LRUtable is cleared and the Best128List is initialized by storing therein a list of 128 pages that have been mapped into the Work Space, but not yet initialized. The GrowWS flag is initially set to False, the ElapsedEpochTime is set to zero and the ElapsedEpochTime timer is started.

VM COMPRESSION MANAGER

15

We will first explain the primary data structures used by the Compression Manager, and then will explain the primary procedures used by the Compression Manager to compress, store and retrieve pages from MappedOut storage.

20

Referring to Figures 9-13, the Compression Manager uses a variable amount of secondary memory to store Mapped Out pages for which there is insufficient room in the Compression Heap 124. When the user accessible virtual memory address space is equal to twice the size of the physical primary memory, it is unusual for any pages to need to be swapped out to secondary memory.

25

However, if most of the contents of the virtual memory address space is highly incompressible, it would be possible for MappedOut storage to occupy an amount of space in the secondary memory that is equal to the size of the physical primary memory. In the preferred embodiment, the Compression Manager initially creates only a small disk file for storing MappedOut pages, and then grows that file as

30

necessary.

- 29 -

Referring to Figure 10, the Sector Table 162 is sized so that it can store one used/unused flag value 230 for each sector of secondary (disk) memory that the MappedOut storage space could potentially occupy. A sector of secondary memory in the preferred embodiment stores 512 bytes of data, and thus 8 sectors
5 would be needed to store a 4096 byte page of uncompressed data. In a system with 8 MBytes of RAM, the Sector Table 162 contains 64K entries 230 for indicating the used/unused status of up to 64K secondary memory sectors.

Every page stored in MappedOut storage is assigned a unique tag value between
10 1 and #VMPages-1, where #VMPages is the maximum number of user accessible virtual memory pages. A Tag value of zero is never assigned to a MappedOut page.

The Main Map.

15 Referring to Figure 11, the Main Map table 172 contains one 6-byte entry 232 for each possible tag value. Each entry 232 in the Main Map 172 includes a 2-byte compression size parameter CompSize, and a 4-byte pointer. The 4-byte pointer normally stores the logical address of the location in the Compression Heap in which the page is stored (actually the location of the end of the first data
20 segment in the Compression Heap in which the page is stored, as will be explained in more detail below). When a page is stored in secondary memory, its Main Map pointer is set equal to -1.

Certain special values of the CompSize parameter are used to indicate
25 MappedOut pages having a special status. In particular, when CompSize = 4096, this means that the MappedOut page is being stored in uncompressed form, (e.g., because the Compressor was unable to compress this page). When CompSize = -1, this means that all 16-bit words in the page are identical in value, in which case the page is not stored in the Compression Heap 124, and instead
30 the value of the repeated 16-bit word is stored in the bottom 2-bytes of the Main Map pointer for that page. Finally, when CompSize is less than or equal to 1024,

- 30 -

this means the page was compressed by a factor of at least 4:1, in which case the page will never be swapped out to the secondary memory.

The Map Chain.

- 5 Referring to Figure 12, the MapChain table 174 stores data indicating the order in which the pages in the Compression Heap were added to MappedOut storage. The MapChain table stores a doubly linked list of MappedOut pages, based on their tag values. In particular, the MapChain table 174 contains one 4-byte entry 234 for each possible tag value. Each entry 234 in the MapChain table 172 includes a 2-byte Back Pointer and a 2-byte Forward Pointer. The Back and Forward Pointers for a particular MapChain entry 234 contain the tag numbers of MappedOut pages before and after the page corresponding to that MapChain entry.
- 10
- 15 Two additional pointers, MapChainStart and MapChainEnd are used to indicate the first and last pages currently included in the MapChain.

- When a page is removed from MappedOut storage or is moved to secondary storage, it is removed from the MapChain table 174 using the RemoveFromMapChain procedure shown in Table 14. In addition, the pointer for the page in the MapChain is set equal to -1 (to indicate the page is stored in secondary memory) and a pointer to the first sector in the secondary memory in which the page is stored is then stored in the MapChain entry for that page. Thus, the pages at the beginning of the MapChain are the pages added to the Compression Heap 124 the longest ago, and thus are the best candidates for being moved to the secondary storage should it become necessary to shift some MappedOut pages to secondary storage.
- 20
 - 25

- 31 -

The Compression Heap.

- Referring to Figure 13, the Compression Heap 124 is organized as follows. The Compression Heap 124 always contains at least two pages of storage space and the pages in the Compression Heap are outside the virtual address space accessible to application programs. The Compression Manager maintains a pointer, EndHeap, to the virtual address location just past the end of the Compression Heap. The Compression Heap is divided into variable size data "segments" 240 (which contain data from pages, usually in compressed form) and Free Space segments 242. Initially, the entire Compression Heap is a single Free Space segment. As pages are added to the Compression Heap, each page is allocated just enough space to store that page (usually in compressed form) in the Compression Heap. Whenever a page in the Compression Heap is released or returned to the Work Space, its space is added to the set of Free Space.
- 15 The set of Free Space segments 242 are maintained as a doubly linked list that is sorted in logical address order. Thus, each Free Space segment 242 in the list is located at a higher logical address than the Free Space segment next lower in the list. Two pointers, FirstFree and LastFree point to the first and last Free Space segments.
- 20 As shown in Figure 13, each Free Space segment contains a 4-byte Back Pointer, to the preceding Free Space segment in the list (if any), a 4-byte Forward Pointer to the next Free Space segment in the list (if any), a 4-byte size parameter that specifies the number of bytes in the Free Space segment, and an end-of segment 4-byte marker equal to 0000 0000 H at the end of the segment. The Back and Forward Pointers in the Free Space segments point to the beginning of other Free Space segments.

- 32 -

The minimum size of a Free Space segment is 16 bytes. If a Free Space segment is 20 bytes or larger, its size parameter is repeated in the 4-bytes preceding the end-of-segment marker.

- 5 Each data segment contains a 4-byte Link at its end, preceded by a 2-byte Size parameter, which indicates the size of the data segment in bytes. When a page is stored as a set of linked data segments in the Compression Heap, the Link at the end of each data segment, other than the last data segment, points to a location 6 bytes before the end of the next data segment for that page. The
- 10 Link of the last data segment for the page stores the 2-byte tag value for the page, for reasons that will be explained below.

The Compression Manager includes the following procedures. GetCompPage 250 is a procedure called by the VM Manager to get a free RAM page to satisfy a page fault. FillFromBottom 252 is a procedure called by GetCompPage 250 when the Compression Heap has enough free space that the Compression Manager can give a free RAM page to the VM Manager, but not enough of that Free Space is located at the end of the Compression Heap to enable the Compression Manager to give the last page in the Compression Heap back to the VM Manager.

- WriteCompPage 254 is a procedure called by the VM Manager to write a page in the Work Space into MappedOut Storage. ReadCompPage 256 is a procedure called by the VM Manager to copy a page in MappedOut storage into a page in the Work Space. ReleaseCompPage 258 is a procedure called by the VM Manager to delete a page from the MappedOut Storage. RemoveFromMapChain 260 is a procedure called by the ReleaseCompPage procedure to delete a specified page from the chain of pages in the MapChain table 174.
- 25 Compress 262 and Decompress 264 are the procedures for compressing a specified page of data and for decompressing a specified page of data, using

- 33 -

a version of the Lempel-Zif data compression technique optimized for execution speed.

SwapOutFlag, FreeHeapBytes, etc..

- 5 The SwapOutFlag is a copy of the last GrowWS flag received from the VM Manager. When the SwapOutFlag is True, the Compression Manager stores MappedOut pages on secondary memory (A) whenever that is necessary to satisfy "GetPage" requests from the VM Manager, and also (B) whenever the VM Manager sends the Compression Manager an incompressible page to
10 compress and store.

FreeHeapBytes is the variable maintained by the Compression Manager to keep track of the number of bytes available in the Compression Heap for storing Mapped Out pages.

15

Get Compressed Page - Compression Manager Procedure

- Referring to Table 9, the GetCompPage procedure has four primary sections. In the first section of GetCompPage, the procedure checks the amount of Free Space in the Compression Heap and the total size of the Compression Heap.
20 The GetCompPage procedure will return a fail signal to the VM Manger (A) if the SwapOutFlag is False and the Compression Heap has less than 2 free pages of Free Space, or (B) if the SwapOutFlag is True and the total size of the Compression Heap is less than three pages (because the Compression Heap must always retain at least two pages of RAM). If either of these failure modes
25 occurs, the GetCompPage procedure returns control to the VM Manager and does not perform the next three sections of the procedure.

- In the second section of the GetCompPage procedure, if the Free Space in the
30 Compression Heap is less than two pages and the SwapOutFlag is True, the GetCompPage procedure successively moves pages from the beginning of the

- 34 -

MapChain to secondary memory until the Free Space in the Compression Heap is greater than two pages (i.e., FreeHeapBytes > 2 x PageSize). To do this, this second section of GetCompPage repeatedly performs the following functions, until Free Space in the Compression Heap is greater than two pages:

- 5 1) Select the first page listed in the MapChain table 174, and determine its size from the CompSize value stored for that page in the Main Map 172.
- 2) Inspect the SectorMap table 162 to find the first contiguous set of available sectors sufficiently large to store the selected page.
- 3) Copy the page to the selected region of secondary memory.
- 10 4) Add the data segments in which the page was stored in the Compression Heap to the list of Free Space segments, updating Free Space segment links and the FirstFree pointer, as necessary. If any new Free Space segment is adjacent a pre-existing Free Space segment, the adjacent new and pre-existing Free Space segments are merged.
- 15 5) Update the Main Map and Map Chain to store in the Map Chain entry for the page a pointer to the first sector in which the page was stored.
- 6) Update the FreeHeapBytes variable to indicate the amount of Free Space in the Compression Heap.
- 20 The second section of GetCompPage, described above, is skipped if the Free Space in the Compression Heap is already greater than two pages.

- After the second section of GetCompPage has been performed or skipped, the third section of GetCompPage checks to see if the size of the last Free Space segment 242 in the Compression Heap is smaller than one page plus 16 bytes - which is the minimum size needed to ensure that the Compression Heap still has a Free Space segment at its end after a page at the end of the Compression Heap is given to VM Manager. If the size of the last Free Space segment 242 is less than this size, the FillFromBottom procedure (which moves data segments from near the end of the heap to the top of the Free Space) is called, repeatedly

- 35 -

if necessary, until the size of the last Free Space segment 242 in the Compression Heap is at least one page plus 16 bytes.

The fourth and last section of GetCompPage computes the logical address of
5 the last page in the Compression Heap (which will be returned to the VM manager), updates the EndHeap pointer by moving it back one page, and decreases the size of the Free Space by subtracting the PageSize from the FreeHeapBytes variable. Then GetCompPage returns control to the VM Manager.

10 **Fill From Bottom - Compression Manager Procedure**

Referring to Table 10 and Figure 13, the FillFromBottom procedure increases the size of the last Free Space segment at the end of the Compression Heap. This is accomplished by performing the following steps:

- 15 1) Find the last data segment in the Compression Heap by looking at the Tag stored at the end of that data segment (which is located 2 bytes before the beginning of the last Free Space segment).
- 2) Using the Tag, look up the location of the first data segment for that page in the Main Map, as well as the size of the page.
- 20 3) Copy the entire page from the Compression Heap 124 into the compression buffer 170.
- 4) Add the data segments in which the page was stored in the Compression Heap to the list of Free Space segments, updating Free Space segment links and the FirstFree pointer, as necessary. For each new Free Space 25 segment that is adjacent a pre-existing Free Space segment, the adjacent new and pre-existing Free Space segments are merged. Update the FreeHeapBytes variable to reflect the data segments added to the Free Space, as well as Free Space created by merging adjacent Free Space segments.
- 5) Copy the page from the compression buffer 170 to the top of the 30 Free Space (starting at the first Free Space segment), updating the FirstFree

- 36 -

pointer and the Free Space segment links to maintain a doubly linked list of Free Space segments in the Compression Heap.

- 6) Update the MainMap, if necessary, to indicate the new Compression Heap location of the page that was moved to the top of the Free Space.
- 5 7) Update the FreeHeapBytes variable to indicate the space occupied by the moved page. Also, if the Free Space segment into which the end of the page was copied was split into a data segment and a Free Space segment, the FreeHeapBytes variable is decreased by 6 bytes to reflect the amount of space occupied by an additional segment link.
- 10 8) Update the LastFree pointer to point to the beginning of the last Free Space Segment (which will now start at a lower address than previously), and then return control to the GetCompPage procedure.

Write Compressed Page - Compression Manager Procedure

- 15 Referring to Table 11, each call to the WriteCompPage procedure 254 includes the following parameters: a page address specified by the calling procedure, the GrowWS flag from the VM Manager, a Tag variable in which the Compression Manager should return the tag assigned to the page being written to MappedOut storage, and a fail flag which the WriteCompPage sets to True under certain circumstances.
- 20

- 25 The WriteCompPage procedure copies the GrowWS flag in the procedure call to the SwapOutFlag and then calls the Compression procedure to compress the data in the page specified by the calling procedure. The Compression procedure stores the compressed page in the Compression Buffer 170 and returns a CompSize parameter indicating the number of bytes occupied by the compressed page.
- 30 If the size of the compressed page indicates that it is incompressible (i.e., the page size was reduced by less than a predefined percentage, such as 12.5

- 37 -

percent) and the SwapOutPage flag is False, the specified page is not stored in MappedOut storage and a fail signal is returned to the calling procedure.

If the page was not incompressible, or the SwapOutPage flag is True, the
5 specified page is assigned a Tag value not used by any other page, and the size (CompSize) of the compressed page is stored in the size field of the associated Main Map entry.

Next, if the page was incompressible, the SwapOutPage flag is True, and the
10 amount of Free Space in the Compression Heap is less than two pages, then the incompressible page is stored directly in secondary memory in uncompressed format, by performing the following steps:

- 1) Inspect the SectorMap table 162 to find the first contiguous set of available sectors sufficiently large to store the selected page.
- 15 2) Copy the uncompressed page to the selected region of secondary memory.
- 3) Store the size of the uncompressed page in the size field of the associated Main Map entry, store a -1 value in the Main Map pointer field, and store in the Map Chain entry for the page a pointer to the first sector in which
20 the page was stored.
- 4) Add the RAM page in which the selected page was stored to the end of the Compression Heap.
- 5) Update the FreeHeapBytes variable by one page to indicate the increased size of the Compression Heap, and update the EndHeap pointer to
25 point to the new end of the Compression Heap.

If the CompSize parameter returned by the Compression procedure is equal to -1, this means that all 16-bit words in the page are identical in value, in which case the page is not stored in the Compression Heap 124, and instead the value
30 of the repeated 16-bit word is stored in the bottom 2-bytes of the Main Map pointer for that page.

- 38 -

- Next, if the CompSize parameter indicates the page was compressible, but not filled with identical words, the compressed page is copied from the Compression Buffer into the Free Space in the Compression Heap. On the other hand, if the CompSize parameter indicates the page was incompressible, and SwapOutPage
5 is False, then the page is copied in uncompressed format from its current RAM page into the Compression Heap and the page size stored in the Main Map entry for this page is reset to be equal to the size of one page. In either case, the location of the page in the Compression Heap is stored in the MainMap entry for the page.
- 10 Next, if the compression ratio for the page stored in the Compression Heap is less than 4:1 (meaning the compressed page is larger than 1024 bytes in the preferred embodiment), the page is added to the end of the Map Chain by adding the appropriate pointers to the corresponding Map Chain entries. In the preferred
15 embodiment of the invention, when a page is compressed by a factor of at least 4:1, it will never be swapped out to the secondary memory because it would be inefficient to do so.
- Finally, the RAM page in which the specified page was stored is added to the
20 end of the Compression Heap, and the FreeHeapBytes variable is updated to indicate the amount of free space in the Compression Heap.
- Read Compressed Page - Compression Manager Procedure**
- 25 Referring to Table 12, the ReadCompPage procedure 256 retrieves a page from MappedOut storage, decompresses it if necessary, and stores the retrieved page in the virtual memory page specified by the calling procedure. The calling procedure specifies both the Tag value of the page to be retrieved, and the virtual memory page into which it is to be copied.
- 30

- 39 -

The procedure starts by reading the page's size and location pointer from the Main Map.

- If the page size is -1, the specified virtual memory page is filled with the 2-byte value stored in the location pointer field, the Main Map entry for the page is updated by setting its size and pointer to 0, and then the procedure exits, returning control to the calling program.

- If the page size indicates that the page was not compressed, then it is copied from either the Compression Heap or Secondary memory, as appropriate, directly to the Transient Page 158 (see Figure 3).

Retrieving Compressed Pages.

- If the page size indicates that the page was compressed, then it is copied from either the Compression Heap or Secondary memory, as appropriate, to the Compression Buffer 170. If the compressed page was stored in secondary memory, the Sector Table is updated to indicate the sectors no longer occupied by stored data.
- Regardless of whether the compressed page was copied from the Compression Heap 124 or the secondary memory, after it is copied into the Compression Buffer 170 the Decompress procedure is called to decompress the page into a full page of data (stored at the Transient Page address).

25 Release Compressed Page - Compression Manager Procedure

Referring to Table 13, the ReleaseCompPage procedure 258 deletes the page specified by the calling procedure from MappedOut storage. The calling procedure specifies the virtual memory address of the page to be released.

30

- 40 -

The procedure starts by determining the tag for the specified page and then reading the specified page's size and location pointer from the Main Map.

If the page's MainMap pointer value is equal to -1, the page is stored on disk.

- 5 In that case, the Sector Table 162 is updated to show as unused those sectors in which the page was stored, and the Map Chain entry for the page is cleared.

- If the specified page is stored in the Compression Heap (and thus does not have a stored size equal to -1) the specified page is deleted by adding the 10 Compression Heap segments in which the page was stored to the set of Free Space segments, updating the Free Space segment links as necessary. If any new Free Space segment is adjacent a pre-existing Free Space segment, the adjacent Free Space segments are merged. Then the FreeHeapBytes variable is updated to indicate the amount of free space in the Compression Heap. In 15 addition, if the page begin released from the Compression Heap has a compressed size that is greater than one fourth a standard page, it is removed from Map Chain by calling the RemoveFromMapChain procedure.

- Details of the RemoveFromMapChain procedure are shown in Table 14, but 20 essentially it is just a matter of removing an item from a doubly linked list and then fixing the forward pointer in the item in the list (if any) before the removed item and the back pointer in the item (if any) after the removed item so that the before and after items are properly linked.

- 25 Finally the Main Map entry for the specified page is updated by setting its size and pointer to 0, and then the procedure exits, returning control to the calling program.

DATA COMPRESSION AND DECOMPRESSION PROCEDURES

- 41 -

Referring to Figures 14, 15 and 16 and Tables 15 and 16, the Compress 262 and Decompress 264 procedures are the procedures for compressing a specified page of data and for decompressing a specified page of data, using a version of the Lempel-Zif data compression technique optimized for execution speed.

5

- The Compression procedure makes use of the LastUse table 166 and Index table 168. Referring to Figure 14, the LastUse table 166 is a table with 64K 2-byte entries, one for every possible value of a 16-bit word. Each page to be compressed contains 2048 16-bit words. As each word of the page is processed, 10 starting with the first word, the position of that word is stored in the LastUse table:

$$\text{LastUse(word value)} = \text{position of word in the page}$$

- where the position value stored in the LastUse table is the position of the first 15 byte of the word (i.e., an even number between 0 and 4094).

- Referring to Figure 15, the IndexTable 168 stores one entry, containing two 2-byte fields, for each of the 2048 words in the page to be compressed. The first field of the entry indicates the most recent position in the page, if any, that the same 20 word was previously encountered. The second field of the entry stores the dictionary index for the word at the page position corresponding to this IndexTable entry.

- An explicit dictionary is not constructed during the data compression procedure, 25 but each time a new unique word is encountered in the page being compressed, it is assigned a new dictionary index value and that value is stored in the Index Table. The first word in the page is always assigned a dictionary index equal to 0. The dictionary for the page can be reconstructed by inspecting the contents of the IndexTable 168 and the page being compressed.

30

- 42 -

Compression Procedure

Referring to Table 15 and Figures 14-16, the compression procedure 262 works as follows. In a first pass through the page to be compressed, the LastUse table
5 entries for all the words in the page are set equal to -1 (box 270 in Figure 16). In addition, if any word is not equal to the preceding word in the page, a flag that was initially set to True is reset to False. Thus, at the end of the first pass through the page, the relevant portion of the LastUse table needed for processing the current page is initialized, and the previously mentioned flag value is True
10 only if all the words in the page are identical.

If all the words in the page to be compressed are identical (box 272), then the compressed size parameter, CompSize, for the page is set equal to -1, and the procedure exits (box 274), returning control to the calling procedure. Otherwise,
15 a second pass through the words in the page is performed.

In the second pass through the page, the contents of the page are compressed, with the resulting compressed data being written into the Compression Buffer 170. As each word of the page is processed, the pointer value in the LastUse
20 table for that word is retrieved

PriorPos = LastUse(word)

and checked to see if it is equal to -1, indicating the word has not previously
25 occurred in the page (box 276). If so, the word is assigned the next available dictionary index (boxes 278, 280, 282), which is stored in the IndexTable::

IndexTable(Position).Index = NextIndex

NextIndex = NextIndex + 1.

- 43 -

Also stored in the IndexTable is a "prior position" value of -1, indicating the word was not previously used in this page:

IndexTable(Position).PriorPos = -1.

5

If the word at the current position did not occur previously in the page, but has a value within a range of ± 1023 of the word four bytes earlier in the page (box 284), the new word is encoded (box 280) as a "new word differential" code:

10 "110" + Dif

where "+" indicates the concatenation of two binary strings, and Dif is the difference between the current word and the word four bytes earlier in the page. Dif is encoded as an eleven bit, signed value. Audio data and many other types
15 of data used in computers have a strong correlation with the stored a fixed number of bytes earlier in memory, and the present invention's compression technique takes advantage of this correlation by preferentially encoding such 16-bit words with a 14-bit encoded value.

20 If the word at the current position did not occur previously in the page, and does not have a value within a range of ± 1023 of the word four bytes earlier in the page, the new word is encoded (box 278) as the following 17-bit "new word" code value:

25 "0" + Word.

Processing for Word Already in the Dictionary.

If the word being processed did previously occur in the current page, the LastUse entry for that word value

30

PriorPos = LastUse(Word))

- 44 -

- will be a pointer to the last previous use of that word in the page. The PriorPos pointer is then used to access the IndexTable entry for that prior use of the same word, and the dictionary index stored in the IndexTable for the prior use of the same word is copied into the dictionary index field for the current word position.
- 5 The PriorPos value obtained from the LastUse table is copied into the "most recent prior use" field of the IndexTable for the current word position, and then the LastUse pointer for the current word is updated to point to the current word position (box 286).
- 10 Next (still box 286), the compression procedure looks for the longest string earlier in the page (but not starting more than 32 occurrences ago of the current word) that matches the string of words starting at the current position in the page. If a matching string of 32 words is found, no further searching for a better match is performed, since the maximum length matching string that can be encoded
- 15 in the compressed data is 32 words long. The length of the best matching string is stored in a variable called BestRun, and the position of the first word in the best matching string is stored in a variable called BestPriorPos.

Best Matching Prior String has a Run Length of 1.

- 20 If the longest prior matching string has a BestRun length of 1, then the current word is encoded as a "repeat one word" code and written to the Compression Buffer (box 288) as:

"10" + EIndex

- 25 where EIndex is a variable length index value defined as follows. If NextIndex is the number of words currently in the Dictionary, then the dictionary index for the current word is encoded using either N or N-1 bits, where N is equal to 1 plus the integer portion of the logarithm, base 2, of NextIndex. In particular, if
- 30 the dictionary index for the current word is less than $2^N - \text{NextIndex}$, then EIndex is equal to the dictionary index encoded as N-1 bits. Otherwise, EIndex is equal

- 45 -

to the dictionary index plus $2^N - \text{NextIndex}$, encoded as N bits. For instance, when the dictionary has 6 entries, the EIndex values for those six dictionary entries are:

	<u>Dictionary Index</u>	<u>EIndex</u>
	0	00
	1	01
	2	100
	3	101
10	4	110
	5	111

- By encoding EIndex with N-1 bits instead of N bits whenever possible, an average of about two-thirds of a bit is saved for each use of the "repeat one word" code.
- 15 Best Matching Prior String has a Run Length Greater than 1.
 If the longest prior matching string has a BestRun length greater than 1, then the current word and the next BestRun-1 words are encoded as a single "repeat prior string" code, using the coding format shown in box 290 of Figure 16B. In
 20 addition, the LastUse and Index table values for all the words in the repeated string are updated.

- After each word or string of words is encoded, the above procedure is repeated starting with the next word of the page (step 292), until all the words in the page
 25 have been processed. The number of bytes of compressed data written to the Compression Buffer 170 is stored in the CompSize parameter returned by the Compression procedure to the calling procedure (step 294).

Decompression Procedure

- 30 Referring to Table 16, the decompression procedure works as follows. The data to be decompressed is located in the Compression Buffer and the decompressed data is written directly into the virtual memory page specified by the calling

- 46 -

procedure. The decompression data comprises a sequence of variable length codes that must be decoded in sequence. The first few bits of each code identify whether the code is a "new word" code ("0"+word), a "new word differential" code ("110"+Dif), a "repeat one word" code ("10"+EIndex), or a "repeat prior string" code ("111"+ BestRun code + DistanceBack code). In addition, decoding the "repeat one word" code and the "repeat prior string" code requires determination of the number of bits used in each such code.

The decompression procedure also generates and stores in the LastUse table a dictionary of words used in the page, with the Xth word in the dictionary being stored in the Xth entry in the LastUse table.

To decompress a compressed page, the decompression procedure sequentially identifies and processes each code in the page. Once a code is identified, if the code is a "new word" or "new word differential" code, the corresponding new word is added to the dictionary stored in the LastUse table and the count (NextIndex) of the number words in the dictionary is increased. The decoded new word is also written to the specified output page. If the identified code is a "repeat one word" code, the specified dictionary entry is written to the specified output page. If the identified code is a "repeat prior string" code, the procedure determines the position of the first word of the string to be repeated and the number of words to be repeated, and then writes a copy of the specified string to the specified output page. Then the next code in the compressed page is processed. This procedure continues until a full page of uncompressed data has been generated, at which point the decompression procedure exits and returns control to the calling procedure.

ALTERNATE EMBODIMENTS

- 47 -

One preferred alternate embodiment of the present invention is to use a co-processor that shares access to the computer's primary and secondary memory to perform much of the VM Manager's work. In particular, when the co-processor is not occupied by other tasks, it could perform the function of

5 periodically rebuilding the Best128List and re-evaluating the WSU flags to determine when the work space needs to grow. Furthermore, whenever the number of free pages in the Compression Heap falls below a threshold number, such as six pages, the co-processor could be programmed to select pages from the Best128List and compress and store them in the Compression Heap until

10 the number of free pages in the Compression Heap reached a target level, such as twelve pages, thereby making sure that a reasonable number of "free pages" are available for use by the VM Manager to satisfy page faults. The co-processor would be assigned its own compression buffer, so that both the co-processor and the regular Compression Manager executed by the primary CPU can operate

15 simultaneously. An additional status flag would be needed in the page table entries to indicate pages currently being processed by the co-processor, and an interlock would be needed so that only one of the two co-processors makes changes to the Page Table at any one point in time.

20 Storing only Decompressed Pages in Secondary Memory

To speed up the fault response time to faults on pages in secondary memory, the present invention could be modified so as to store only uncompressed pages in the secondary memory. Thus, when a page is moved from the Compression

25 Heap to secondary memory, it would first be decompressed and then stored in secondary memory. While this would increase the amount of secondary memory used, it would not increase the number of times the secondary memory is accessed, and it would decrease the fault response time for faults on pages in secondary memory.

30

- 48 -

While the present invention has been described with reference to a few specific embodiments, the description is illustrative of the invention and is not to be construed as limiting the invention. Various modifications may occur to those skilled in the art without departing from the true spirit and scope of the invention
5 as defined by the appended claims.

TABLE 1

© Copyright 1993 Connectix Corporation

5 **Top Level Procedure for Processing a Virtual Memory Fault**

ReceiveFault(page#,tag,flags) -- VMU Fault signal received

10 **– (A) Get Page to Satisfy the Page Fault**
 If (MI=0 and MO=0) -- normally means the faulting virtual address
 is currently unused

15 {
 Call GetPage() -- Get physical RAM page to satisfy fault
 If MD=0 -- MI,MD,MO=0,1,0 is a special marking for allocated
 page that will be overwritten by data from disk.
 Thus, if MD=1, we don't bother initializing the page.

20 {
 -- Fill page with highly compressible data
 Call FillPage(TransientPageAdr/PageSize)
 }
 }
 Elseif (MI=0 and MO=1) -- Page is located in compressed storage
 (either in RAM or on disk)

25 {
 Call GetPage() -- Get physical RAM page to satisfy fault
 -- Retrieve page identified by "tag" from MappedOut Storage and copy
 it into the allocated RAM page at TransientPageAdr
 Call ReadCompPage(tag)
 Call ReleasePage(page#)
 }

30 -- Map the received page into the specified virtual memory page
 Call MapPage(page#)

35 **– (B) Check for Foreground Context Switch**
 If Foreground Process Context has changed since the last page fault
 -- Clear the WSU bit flags used for determining if more pages are
 needed in WorkSpace
 { Clear WorkSpaceUsed (WSU) bit flag in all VMPageMap entries}

40 **– (C) Periodically Update the WorkSpaceUsed (WSU) Flags**
 If ElapsedEpochTime > EpochThreshold -- Epoch Threshold is nominally
 3 seconds.

45 {
 Clear WorkSpaceUsed flag (WSU) in all VMPageMap entries
 Restart ElaspedEpochTime timer at ElapsedEpochTime = 0
 }
 Return

- 50 -
TABLE 2

Procedures for Getting a RAM Page to Use

```

5   GetPage()
    Set Flag1 = False
    Do Until (Flag1)
        - Repeat this Loop until the
          Compression Manager sends a RAM
          page to use

10  {
    -- Ask Compression Manager to send one unused RAM page
    Call GetCompPage(PageAdr,fail)
    If fail=True
        -- Compression Manager did not have an
        unused RAM page to send
    15 {Call SwapOutPage(PageAdr)} -- Swap one page from RAM to
        MappedOut Storage
    If PageAdr ≠ 0
        -- Compression Manager did have an unused
        RAM page to send, or the SwapOutPage
        procedure found an uninitialized page to use

20  {
    -- Map the received page into the TransientPageAdr in the VMU
    page table:
    Set Flag1 = True
    page#           =TransientPageAdr/PageSize
25  RecvdPage#     =PageAdr/PageSize
    RAM_Adr         =VMPageMap(RecvdPage#).Adr
    VMPageMap(page#).Adr =RAM_Adr
    VMPageMap(page#).MI  =1      -- Set Mapped In Flag
    -- Leave Modified Flag (MD) unchanged
    VMPageMap(page#).MO  =0      -- Reset Mapped Out Flag

30
    -- Un-map the received page from its prior virtual address
    VMPageMap(RecvdPage#).Adr =0
    VMPageMap(RecvdPage#).MI  =0
35  VMPageMap(RecvdPage#).MD =0
    VMPageMap(RecvdPage#).MO  =0

35  }
    -- end of Do Until Loop
40  Return

FillPage(Page#) -- Fill page with highly compressible data (could use any
                  fill value that is repeated for all words in the page)
45  Fill all words in specified virtual address page with "0000"
    Return

```

```

MapPage(Page#)
  -- Map the received page from the TransientPageAdr into the VMU page
  table:
    RecvdPage#           =TransientPageAdr/PageSize
    RAM_Adr              =VMPageMap(RecvPage#).Adr
    VMPageMap(page#).Adr =RAM_Adr
    VMPageMap(page#).MI  =1      -- Set Mapped In Flag
    -- Leave Modified Flag (MD) unchanged
    VMPageMap(page#).MO  =0      -- Reset Mapped Out Flag
    VMPageMap(page#).U   =0      -- Reset Used Flag
    VMPageMap(page#).WSU =0      -- Reset WorkSpaceUsed Flag

  -- Update Count of Uncompressed Pages in WorkSpace
  #UncompressedPages = #UncompressedPages + 1

Return

```

TABLE 3

Procedure for Swapping Out a Page

5 SwapOutPage(PageAdr)

- (A) Preprocessing before each Page Swap Out:

PageAdr = 0 – Normal return value

StartTime = ElapsedEpochTime – Store ElaspedEpochTime timer value for restoration at end of swapping procedure

- LRUtable represents usage history for period in which the number of page swaps equals the number of swappable pages in the Work Space

#Swaps = #Swaps + 1

SwapThreshold = #UncompressedPages / 16

If #Swaps > SwapThreshold

```
{  
    Call AgingEvent()           -- Updates LRU data table  
    #Swaps = 0  
}
```

– (B) Swap Out Control Loop

Set Flag2 = False

PG = 0

Do Until (Flag2) – REPEAT until a page is swapped out

1

(B-1) SELECT A PAGE TO BE SWAPPED OUT

If GrowWS = WorkSpace needs to grow

1

PG = Logical address of page highest on Best128List with MI=1 and U=0

Else -- WorkSpace does not need to grow

PG = Logical address of page highest on Best128List which is Mapped In (MI=1) and not currently used (U=0), but which does not have MI,MD,MO=1,0,1 (— a special marker for incompressible pages (compressed less than threshold, e.g., 12.5%)). Don't bother trying to send an incompressible page to the Compression Manager unless GrowWS is True

- None of the pages in the Best128List are eligible for being swapped out! Rebuild the Best128List:

Call BuildBest128List() -- Rebuilds Best128 list and updates GrowWS flag

- 53 -

```

If GrowWS -- WorkSpace needs to grow
{
    PG = Logical address of page highest on Best128 list with
          MI=1 and U=0
}
Else      -- WorkSpace does not need to grow
{
    PG = Logical address of page highest on Best128 list which
          is Mapped In (MI=1) and not currently used (U=0), but
          which does not have MI,MD,MO=1,0,1
}

If PG = 0  -- Still no swappable pages in the Best128 list.
{
    -- (all pages in Best128 list are incompressible !)
Do Until PG ≠ 0
{
    Call AgingEvent()
    PG = Logical address of page highest on Best128 list
          which is Mapped In (MI=1) and not currently used
          (U=0), but which does not have MI,MD,MO=1,0,1
}
}

-- (B-2) SWAP OUT THE SELECTED PAGE
-- Reset selected page's usage data to FFFF
RAMPage# = VMPageMAP(PG).Adr
LRUtable(RAMPage#) = FFFF

-- If selected page is an uninitialized page, just return it and quit.
If VMPageMap(PG).MI,MD,MO=1,0,0
{
    PageAdr = PG
    -- Restore saved ElapsedEpochTime timer value
Set ElapsedEpochTime = StartTime
Return
}

-- send page PG to Compression Manager for compression
Call WriteCompPage(PG,GrowWS,Tag,fail)

If fail=True      -- Page was incompressible (compressed less
                  than threshold, e.g., 12.5%)
{
    -- Set VM Page Table Flags with special marker
    so that MI,MD,MO = 1,0,1
    Page# = PG / PageSize
    VMPageMap(Page#).MD = 0 -- Reset Modified Flag
    VMPageMap(Page#).MO = 1 -- Set Mapped Out Flag
}

```

- 54 -

```
    }
Else
{
  -- Update VMPageMap for page written to MappedOut Storage
  5   Page# = PG / PageSize
      VMPageMap(Page#).Adr = Tag
      VMPageMap(Page#).MI = 0
      VMPageMap(Page#).MD = 0
      VMPageMap(Page#).MO = 1
  10  -- Update Count of Uncompressed Pages in WorkSpace
      #UncompressedPages = #UncompressedPages - 1

  15  Flag2 = True -- Page at PG has been successfully swapped out
      }

  } -- end of Do Until Loop
- Restore saved ElapsedEpochTime timer value
  Set ElapsedEpochTime = StartTime
  Return
```

- 55 -

TABLE 4

Procedure for Updating LRU Page Usage Data Table

```

5      AgingEvent()
       -- Simultaneously Update Page Usage Data and Rebuild Best128 List
10     -- Clear Best128 List
          BestAge = FFFF H
          #BestEntries = 0
          #PagesUsed = 0
15     -- Do Just One Pass through the VMPageMap
          XP = StartSweepPtr
          StartSweepPtr = Ptr + 1 Modulo #VMPages
          For Page# = XP to #VMPages and then For Page# = 0 to XP-1
          {
             -- Ignore UnSwappable Pages, Pages in MappedOut Storage, and
20           uninitialized non-resident pages
             If ( VMPageMap(Page#).NS = 0
                  .and. VMPage(Page#).MI,MD,MO ≠ 0,0,1
                  .and. VMPage(Page#).MI,MD,MO ≠ 0,0,0 )
                  {
                     -- Shift Usage Data into LRUtable
                     RAMPage# = VMPageMAP(Page#).Addr
                     RightShift VMPageMAP(Page#).U into LRUtable(RAMPage#)
                     -- Update WorkSpace Usage bits
30           VMPageMap(Page#).WSU = VMPageMap(Page#).WSU
                           .or. VMPageMap(Page#).U
                     -- Count Number of Swappable Pages with WSU=1
                     If VMPageMap(Page#).WSU
                         { #PagesUsed = #PagesUsed + 1 }

                     -- Clear Used Bit Flag
                     VMPageMap(Page#).U = 0
40           -- Update Best128List
                     If ( LRUtable(RAMPage#) = BestAge and #BestEntries < 128 )
                         {
                            -- Add page to Best128List
                            Best128List(#BestEntries) = Page#
                            #BestEntries = #BestEntries + 1
                         }
45           If LRUtable(RAMPage#) < BestAge
                         {

```

- 56 -

```
-- Clear Best128List and Restart with current Page#
BestAge = LRUtable(RAMPage#)
Best128List(0) = Page#
#BestEntries = 1
5      }
}
) -- End of For Loop

10     --
11     - Update GrowWS Flag
12     -
13     - Note that #PagesUsed equals the number of Pages in VMPageMap with
14       WSU = 1, excluding UnSwappable Pages (NS=1) and excluding Pages
15       whose Page# > #VMPages

16     If #PagesUsed > #UncompressedPages
17       { Set GrowWS = True }
18     Else
19       { Set GrowWS = False }

20     Return
```

- 57 -

TABLE 5

Procedure for Building Best128List - List of Best Pages to Swap Out

```

5      BuildBest128List()
      -- Similar to AgingEvent() procedure, except that Page Usage Data is not
      -- updated

10     -- Clear Best128 List
      BestAge = FFFF H
      #BestEntries = 0
      #PagesUsed = 0

15     -- Do Just One Pass through the VMPageMap
      XP = StartSweepPtr
      StartSweepPtr = Ptr + 1 Modulo #VMPages
      For Page# = XP to #VMPages and then For Page# = 0 to XP-1
      {
      -- Ignore UnSwappable Pages, Pages in MappedOut Storage, and
      -- uninitialized non-resident pages
      If ( VMPageMap(Page#).NS = 0
           .and. VMPageMap(Page#).MI,MD,MO ≠ 0,0,1
           .and. VMPageMap(Page#).MI,MD,MO ≠ 0,0,0 )
      {
      -- Update WorkSpace Usage bits
      VMPageMap(Page#).WSU = VMPageMap(Page#).WSU
           .or. VMPageMap(Page#).U

20
25
30
35
40
45
      -- Count Number of Swappable Pages with WSU=1
      If VMPageMap(Page#).WSU
           { #PagesUsed = #PagesUsed + 1 }

      -- Update Best128List
      If (VMPageMap(Page#).U = 0) -- Don't put pages currently in
           use into the Best128List
           {
           If ( LRUtable(Page#) = BestAge .and. #BestEntries < 128 )
           {
           -- Add page to Best128List
           Best128List(#BestEntries) = Page#
           #BestEntries = #BestEntries + 1
           }
           If LRUtable(Page#) < BestAge
           {
           -- Clear Best128List and Restart with current Page#
           BestAge = LRUtable(Page#)
           Best128List(0) = Page#
           }
      }
  
```

- 58 -

```
        #BestEntries = 1
    }
}
5      }  -- End of For Loop

- Special Handling If All Pages are In Use
If #BestEntries = 0
    { Call AgingEvent() }
10     Else
- Update GrowWS Flag
- Note that #PagesUsed equals the number of Pages in VMPageMap with
  WSU = 1, excluding UnSwappable Pages (NS=1) and excluding Pages
15     whose Page# > #VMPages
{
    If #PagesUsed > #UncompressedPages
        { Set GrowWS = True }
    Else
        { Set GrowWS = False }
20
}
Return
```

- 59 -

TABLE 6

Procedure For Handling Read Operation

5 **Read(first,last)**
 -- An I/O Read from disk or elsewhere is filling all pages from first to last
 For Page# from first to last
 {
 -- Special Handling for pages in MappedOut Storage
10 If VMPageMap(Page#).MI=0 and VMPageMap(Page#,MO)=1
 {
 -- Compression Manager releases space allocated to this page
 Call ReleaseCompPage(Page#)
 -- Set Up VMPageMap
 Clear Flags MI and MO of VMPageMap(Page#)
 Set Flag MD of VMPageMap(Page#) -- special marker
 }
 If VMPageMap(Page#).MI = 1
20 {
 -- keep as resident page, but do not bother to initialize
 Clear Flags MD and MO of VMPageMap(Page#)
 }
 }
25 } -- End of For Loop
Call Standard OS Read() procedure
Return

- 60 -

TABLE 7

Procedures For Handling Memory Range and Page Deallocation

```

5     DeallocateMem(begin,end)      -- This is called when a program deallocates
          memory from addresses "begin" to "end"
          If (the address range "begin" to "end" includes any complete pages)
          {
            first = address of the first complete page in the specified address range
            last = address of the last complete page in the specified address range
            Call DeallocatePages(first,last)
          }
          Fill the deallocated portions of any partially deallocated pages with zeros
          Return
15

DeallocatePages(first,last)      -- This is called when a program terminates
                                or deallocate pages no longer needed
          For Page# from first to last
          {
            -- Special Handling for pages in MappedOut Storage
            If VMPageMap(Page#).MI=0 and VMPageMap(Page#).MO=1
            {
              -- Compression Manager releases space allocated to this page
              Call ReleaseCompPage(Page#)
              -- Set Up VMPageMap
              Clear Flags MO, MD, U and WSU of VMPageMap(Page#)
            }
30          If VMPageMap(Page#).MI = 1
            {
              -- Keep as resident page, but ...
              -- The Modified (MD) flag is cleared so that this deallocated logical
                 page will not be saved to MappedOut Storage
35          Clear Flags MD, MO, U and WSU of VMPageMap(Page#)
              Call FillPage(page#)    -- Fill page with highly compressible data
              -- Set Usage History to 0000 H so that the RAM page can be
                 reused
40          Clear LRUtable(Page#)
          }
          } -- End of For Loop
          Return

```

Table 8**VM Manager Initialization**

- 5 **Initialize()**
 - This procedure is called during standard OS boot sequence
- 10 Extend the VMPageMap to include pages for addresses through 2xPhysMem and addresses 40000000 through 40000000 + 2xPhysMem
- 15 Install VM Manager, VM Compression Manager, Compression Manager Tables
 - Assign initial set of RAM pages to Compression Heap
- 20 Assign all remaining RAM pages to the next logical addresses to be assigned by the OS.
 - In multiprocessing implementations, the remaining RAM pages would be added to the end of the Compression Heap, which acts as a RAM page free list.
- 25 Initialize VMPageMap to indicate UnSwappable pages for VM Manager and Compression Manager and to initialize Used and WorkSpaceUsed flags
 - Initialize Best128List
 - Set GrowWS = False
 - Set ElapsedEpochTime = 0
 - Start ElapsedEpochTime timer
- 30 Return

- 62 -

Table 9

COMPRESSION MANAGER PROCEDURE
Get Free Page from Compression Heap

5

GetCompPage(PageAdr,fail)

- **(A) Check Amount of Free Space In Heap**
- If Free Space is < 2 Pages, Compression Manager cannot give a free page to the VM Manager.
- 10 If ((FreeHeapBytes ≤ 2 x PageSize) .and. Not(SwapOutFlag))
 - {
 - fail=True
 - Return
 - }
- 15 - If the total size of the Compression Heap is less than 3 pages, this means the VM Manager probably has unused pages of RAM mapped into unused virtual memory locations, and therefore a Fail signal is sent to the VM Manager, which will force it to find old unused pages to send to the Compression manager.
- 20

- 25 **HeapSize = EndHeap - BeginHeap**
If HeapSize < 3 x PageSize
 - {
 - fail=True
 - Return
 - }
- 30 - **(B) If SwapOutFlag is True and Free Space < 2 pages, Generate Free Space by Moving Pages from Compression Heap to Secondary Memory.**
If ((FreeHeapBytes ≤ 2 x PageSize) .and. SwapOutFlag)
 - {
 - Do Until (FreeHeapBytes > 2 x PageSize)
 - {
 - Tag = MapChainStart
 - Find SectorPtr to first sector in a sequence of free sectors sufficiently large to store page PG
 - Copy Page associated with Tag from Compression Heap into Secondary Memory starting at sector specified by SectorPtr.
 - 40 Add segments in which the moved Page was stored to set of Free Space segments, updating Free Space segment links and FirstFree pointer, as necessary. If any new Free Space segment is adjacent

- 63 -

a pre-existing Free Space segment, merge the adjacent new and pre-existing Free Space segments.

5 -- Store Relevant Bookkeeping Data
 Update SectorMap to indicate sectors occupied by swapped out page
 MapChainStart = MapChain(Tag).ForwardPtr
 MapChain(Tag) = SectorPtr
 MainMap(Tag).Ptr = -1

10 FreeHeapBytes = FreeHeapBytes + MainMap(Tag).Size
 + 6 x (Number of segments merged with Free
 Space segments)
 } -- End of Do Loop
 }

15 -- (C) If Size of the Last Free Block is < 1 Page + 16 Bytes, move
 compressed data from near the end of the Compression Heap to
 the Free Space closest to the top of the Compression Heap until
 the Last Free Block is > 1 page

20 fail=False
 Flag3 = False
 Do Until (Flag3)
 {
 SizeLastFreeBlock = EndHeap - LastFree
 If SizeLastFreeBlock < PageSize + 16 Bytes
 { Call FillFromBottom() } -- Move One Compressed Page
 toward front of Compression Heap
 Else
 { Flag3 = True}
 } -- End of Do Loop

25 -- (D) Provide Free Page at end of Compression Heap to VM Manager
 -- Compute 32-bit Logical Page Address of Free Page
 PageAdr = EndHeap - PageSize

30 -- Update EndHeap and FreeHeapBytes
 EndHeap = EndHeap - PageSize
 FreeHeapBytes = FreeHeapBytes - PageSize

35 Return

TABLE 10
COMPRESSION MANAGER PROCEDURE

5

Fill Heap From Bottom Procedure

- FillFromBottom()**
 - Find Last Page in Compression Heap
 - LastTag = RAM(FirstFree - 2)
 - 10 HeapLocation = MainMap(LastTag).Ptr
 - CompSize = MainMap(LastTag).Size
 - Copy Last Page to Compression Buffer
 - 15 Copy entire Page from Compression Heap, starting at HeapLocation, into Compression Buffer.
 - Add segments in which the Last Page was stored to set of Free Space segments, updating Free Space segment links and FirstFree pointer, as necessary. For each new Free Space segment that is adjacent a pre-existing Free Space segment, merge the adjacent new and pre-existing Free Space segments.
 - Update FreeHeapBytes
 - 25 FreeHeapBytes = FreeHeapBytes + CompSize +
 6 x (Number of segments merged with Free Space segments)
 - Copy Last Page to Top of Free Space
 - 30 Copy Page in Compression Buffer into Free Space (starting at First Free Space segment) of Compression Heap (starting at First Free Space segment), updating FirstFree and Free Space segment links to maintain a doubly linked list of Free Space segments in the compression heap.
 - Update MainMap, if necessary, to indicate new Compression Heap location of the page that was moved to the top of the Free Space.
 - 35 -- Update FreeHeapBytes
 - FreeHeapBytes = FreeHeapBytes - CompSize
 - If (the Free Space segment into which the end of the Page was copied was split into a data segment and a Free Space segment)
 - { FreeHeapBytes = FreeHeapBytes - 6 }
 - 40 Update LastFree to point to beginning of last Free Space Segment
 - Return

Table 11

COMPRESSION MANAGER PROCEDURE

5 Write Page into MappedOut Storage Space

```

WriteCompPage(PG,GrowWS,Tag,fail)

10          -- Update Compression Manager's SwapOut Flag
              (for use by GetCompPage procedure)
SwapOutFlag = GrowWS
fail = False

15          -- Call Compression Procedure
Call Compress(PG,CompSize)

              -- Fail if SwapOutFlag is False and page is incompressible
If (CompSize > Threshold -- e.g., 87.5% of PageSize, or 3584 Bytes
20          .and. Not(SwapOutFlag))
{
    fail = True        -- Tell the VM Manager that page is incompressible
    Return
}

25          -- Store CompSize in Main Map and then Update FreeMapEntry
Tag = FreeMapEntry
MainMap(Tag).Size = CompSize
FreeMapEntry = smallest TagValue for which MainMap(TagValue).Size=0

30          -- Store Incompressible Page In Secondary Memory
-- Check to see if Page at PG should be swapped out to secondary memory
If ( GrowWS .and. (FreeHeapBytes ≤ 2 x PageSize)
35          .and. CompSize > Threshold )
{
    Find SectorPtr to first sector in a sequence of free sectors sufficiently
    large to store page at PG

40          Store Uncompressed Page from page at PG into Secondary Memory
              starting at sector specified by SectorPtr.
              CompSize = PageSize

              -- Store Relevant Bookkeeping Data in MainMap and MapChain
45          MainMap(Tag).Size = PageSize
              MainMap(Tag).Ptr = -1        -- indicates page is in
              MapChain(Tag) = SectorPtr    -- Secondary Memory Address

```

- 66 -

```

-- Add Page at PG to end of Compression Heap
Page# = PG / PageSize
VMPageMap(NextCompHeapPage).Adr = VMPageMap(Page#).Adr
VMPageMap(NextCompHeapPage).MI = 1
VMPageMap(NextCompHeapPage).MO = 0
-- Leave Modified (MD) flag unchanged

-- Update FreeHeapBytes and NextCompHeapPage
FreeHeapBytes = FreeHeapBytes + PageSize
NextCompHeapPage = NextCompHeapPage + 1
Return
}

-- Store Page In Compression Heap
If CompSize ≤ Threshold -- e.g., 87.5% of PageSize, or 3584 Bytes
{
  If CompSize = -1 -- -1 indicates page filled with identical words
    -- Special Handling for Page filled with identical words.
    -- Word Value is stored in MainMap(Tag).Ptr instead of in the
      Compression Heap
    { MainMap(Tag).Ptr = RAM(PG) }
  Else
    {
      Copy Compressed Page in Compression Buffer into Free Space of
      Compression Heap, updating FirstFree and Free Space segment
      links as necessary
    }
}
Else -- Store Page in Uncompressed Format into Compression Heap
{
  Copy Uncompressed Page from page at PG into Free Space of
  Compression Heap, updating FirstFree and Free Space segment links
  as necessary
  CompSize = PageSize
  MainMap(Tag).Size = PageSize
}

-- Store Relevant Bookkeeping Data in MainMap and MapChain
MainMap(Tag).Ptr = Pointer to end of the first data segment in which the
compressed page was stored.

-- Add compressed page to Map Chain, unless compression ratio ≥ 4
If CompSize > 0.25 x PageSize
{
  MapChain(Tag).BackPtr = MapChainEnd
  MapChain(Tag).ForwardPtr = 0 -- Tag = 0 is invalid value
  MapChain(MapChainEnd).ForwardPtr = Tag
  MapChainEnd = Tag
}

```

- 67 -

}

-- Add Page PG to end of Compression Heap
Page# = PG / PageSize
5 VMPageMap(NextCompHeapPage).Adr = VMPageMap(Page#).Adr
 VMPageMap(NextCompHeapPage).MI = 1
 VMPageMap(NextCompHeapPage).MO = 0
-- Leave Modified (MD) flag unchanged

10 -- Update FreeHeapBytes
 If CompSize = -1
 { FreeHeapBytes = FreeHeapBytes + PageSize }
 Else
 {
15 FreeHeapBytes = FreeHeapBytes + PageSize - CompSize
 If (the Free Space segment into which the end of the Page was copied
 was split into a data segment and a Free Space segment)
 { FreeHeapBytes = FreeHeapBytes - 6 }
 }
20 Return

Table 12

COMPRESSION MANAGER PROCEDURE

5 Retrieve and Decompress Page from MappedOut Storage

```

ReadCompPage(Tag)
  - Locate the Page In Compression Heap or Secondary Memory
10   HeapLocation = MainMap(Tag).Ptr
      CompSize    = MainMap(Tag).Size
      If HeapLocation = -1
          { SectorPtr = MapChain(tag) }

15   - Special Handling for Page filled with Identical words.
      If CompSize= -1 -- -1 indicates page filled with identical words
      -- Word Value is stored in MainMap(Tag).Ptr instead of Heap Location
      {
          Word = MainMap(Tag).Ptr
          Fill Page at TransientPageAdr with Word
          Return
      }

20   - If page not compressed, copy Page into RAM starting at the Logical
      Address corresponding to Page#
      If CompSize = PageSize
          {
              If HeapLocation = -1
                  {
                      Copy Uncompressed Page from Secondary Memory, starting at
                      sector specified by SectorPtr, into page at TransientPageAdr
                  }
              Else -- Uncompressed page is in Compression Heap
                  {
                      Copy Uncompressed Page from Compression Heap, starting at
                      HeapLocation, into page at TransientPageAdr
                  }
          }

25   Else
      - If Page is compressed, copy compressed Page data into
          Compression Buffer, then decompress the page and write the
          decompressed data into RAM starting at the Logical Address
          corresponding to Page#
      {

40   If HeapLocation = -1
      {

45   }
}

```

- 69 -

```
Copy Compressed Page from Secondary Memory, starting at sector  
specified by SectorPtr, into Compression Buffer.  
}  
Else -- Page is in Compression Heap  
{  
    Copy Compressed Page from Compression Heap, starting at  
    HeapLocation, into Compression Buffer.  
  
    -- Decompress page in Compression Buffer and store decompressed  
    page at TransientPageAdr  
    Call Decompress(CompSize)  
}  
  
Return
```

- 70 -
Table 13

Release Page in MappedOut Storage

```

5      -- Compression Manager releases space allocated to this page
      ReleaseCompPage(Page#)

      -- Locate the Page In Compression Heap or Secondary Memory
      Tag = VMPageMap(Page#).Adr
      10     HeapLocation = MainMap(Tag).Ptr
              CompSize = MainMap(Tag).Size
              If HeapLocation = -1
                  { SectorPtr = MapChain(tag) }

      15     -- If page is in Secondary Memory, release the sectors assigned to
              this page
              If HeapLocation = -1
                  {
                      Update SectorTable to show as unused those sectors in which Page
                      was stored.
                      MapChain(tag) = 0 -- Clear Disk Address in MapChain
                  }
              Elseif CompSize ≠ -1 -- If page is in Compression Heap, add all
              25     segments for this page to the Free Space
                      of Compression Heap, updating Free Space
                      segment links as necessary
                  {
                      Add segments in which page was stored to set of Free Space segments,
                      updating Free Space segment links as necessary. If any new Free
                      Space segment is adjacent a pre-existing Free Space segment, merge
                      the adjacent new and pre-existing Free Space segments.

                      -- Update FreeHeapBytes
                      FreeHeapBytes = FreeHeapBytes + CompSize
                      35     + 6 x (Number of segments merged with Free
                             Space segments)
                      If CompSize > 0.25 x PageSize
                          { Call RemoveFromMapChain(Tag) }
                  }
              40     -- Update Bookkeeping Entries in MainMap
              MainMap(Tag).Ptr = 0
              MainMap(Tag).Size = 0
              FreeMapEntry = smallest TagValue (>0) for which
              45     MainMap(TagValue).Size = 0

```

Return

- 71 -
Table 14

Procedure for Removing a Tag from the MapChain

```
5      RemoveFromMapChain(Tag)
        -- Special Handling if this is first entry in MapChain
        If (MapChainStart = Tag)
            {
                If (MapChainEnd = Tag)    -- Is this only item in MapChain?
                    {
                        MapChainStart = 0
                        MapChainEnd = 0
                    }
                Else
                    { MapChainStart = MapChain(Tag).ForwardPtr }
            }

        -- Special Handling if this is last entry in MapChain
        If (MapChainEnd = Tag)
            { MapChainEnd = MapChain(Tag).BackPtr }

        -- Relink MapChain entries before and after released page, unless page
        was in Secondary Memory
        If ( Tag ≠ MapChainStart .and. Tag ≠ MapChainEnd
            .and. MainMap(Tag).Ptr ≠ -1 )
            {
                Temp1 = MapChain(Tag).BackPtr
                Temp2 = MapChain(Tag).ForwardPtr
                MapChain(Temp1).ForwardPtr = Temp2
                MapChain(Temp2).BackPtr = Temp1
            }
        Return
```

- 72 -
Table 15

Compression Procedure

```

5   Compress(Page#,CompSize)
    -- Initialize Dictionary Table LastUse
    Base = Page# x PageSize
    Word = RAM(Base)
    LastUse(Word) = -1
10   Flag4 = True

    For Position = 1 to PageSize - 2      -- positions are byte positions
    {
        PriorWord = Word
15      Word = RAM(Base + Position)
        If Word ≠ PriorWord
            { Flag4 = False }
        Else
            { LastUse(Word) = -1 }
20      }

    -- Special Handling for Page filled with identical words.
    If Flag4 = True
    {
25      CompSize = -1
        Return
    }

    -- Process first Word (which will be first word in the Dictionary)
30      Word = RAM(Base)
        LastUse(Word) = 0
        IndexTable(0).Index = 0
        IndexTable(0).PriorPos = -1
        NextIndex = 1
35      Write to CompressBuffer "0"+Word
        CompSize = 17 (bits)

    -- Process Page in units of 2-Byte words
    Position = 2
40      Do Until Position ≥ PageSize - 2
    {
        Word = RAM(Base + Position)

        -- Is this first time Word has been Seen?
        If LastUse(Word) = -1
        {
            -- Processing for Word Not Already in the Dictionary
            LastUse(Word) = Position
        }
    }

```

- 73 -

```

IndexTable(Position/2).Index = NextIndex
IndexTable(Position/2).PriorPos = -1
NextIndex = NextIndex + 1

5      -- Special Handling for Audio Data and the like:
Dif = Word - RAM(Base + Position - 4)
If ( (Position > 2) .and. (|Dif| < 1024) )
{
    Write to CompressBuffer "110"+  

        Dif(encoded as 11-bit signed value)
    CompSize = CompSize + 14
}
Else
{
    Write to CompressBuffer "0"+Word
    CompSize = CompSize + 17
}
Position = Position + 2
}
20     Else
{
    -- Processing for Word Already in the Dictionary
    PriorPos = LastUse(Word)
    IndexTable(Position/2).PriorPos = PriorPos
25     IndexTable(Position/2).Index = IndexTable(PriorPos/2).Index
    LastUse(Word) = Position

    BestPriorPos = PriorPos
    BestRun = 1
30     #Tries = 0

    Do Until ( (BestRun = 32) .or. (PriorPos=-1) .or. (#Tries=32) )
    {
        #Tries = #Tries + 1
        35     RunLength = Number of words in Page, starting at PriorPos,
                which match words starting at the current
                Position, not to exceed 32)
        If RunLength > BestRun
        {
            BestRun = RunLength
            BestPriorPos = PriorPos
        }
        PriorPos = IndexTable(PriorPos/2).PriorPos
    } -- End of Do Loop
45     If BestRun = 1      -- Best match = 1 word long:
    {
        DIndex = IndexTable(Position/2).Index
    }
}

```

- 74 -

```

N = 1 + IntegerPortion( Log2(NextIndex) )
Boundary = 2N - NextIndex
If DIndex ≥ Boundary
    { EIndex = (DIndex + Boundary) encoded as N bits }
5      Else
            {
                N = N - 1
                EIndex = DIndex encoded as N bits
            }
10     Write to CompressBuffer "10"+ EIndex
        CompSize = CompSize + 2 + N
        Position = Position + 2
    }
15     Else — Encoding Procedure for best match is > 1 word long:
            {
                Update LastUse and IndexTable entries
                for words at Position+2 through Position+(2x(BestRun-1))

                BeginCase (BestRun)
20                    Case (BestRun=2)
                        { E.Length = "0" }
                    Case (BestRun=3)
                        { E.Length = "10" }
                    Case (BestRun=4)
                        { E.Length = "110" }
                    Case (BestRun > 4)
                        {
                            Value = BestRun - 2
                            E.Length = "111" + Encode(Value,5 bits,unsigned)
                        }
30                    EndCase

                    DistBack = Position - BestPriorPos
                    BeginCase (DistBack)
35                    Case (DistBack ≤ 16)
                        {
                            E.Distnc = "00" + Encode(DistBack - 1, 4 bits, unsigned)
                        }
                    Case (16 < DistBack ≤ 80)
                        {
                            E.Distnc = "01" + Encode(DistBack - 17, 6 bits, unsigned)
                        }
                    Case (80 < DistBack)
                        {
                            E.Distnc = "1" + Encode(DistBack, 11 bits, unsigned)
                        }
40                    EndCase
45
    
```

- 75 -

Write to CompressBuffer "111"+ E.Length + E.Distnc
CompSize = CompSize + 3 + (bit length of E.Length and E.Distnc)

Position = Position + (BestRun x 2)

5

}

}

} — End of Main Loop

-- Convert CompSize to number of bytes written to CompressBuffer

CompSize = RoundUp(CompSize / 8)

10 Return

- 76 -
Table 16

Decompression Procedure

5

Decompress(CompSize) -- Decompress page in Compression Buffer and store decompressed page in the Transient Page at TransientPageAdr

10

- IndexTable is not used during Decompression
- LastUse table does not need to be initialized
- LastUse table is used as a Dictionary during Decompression

15

```
Base = TransientPageAdr
BitPos = PtrCompBuffer
NextIndex = 0
```

-- Process first Word (which will be first word in the Dictionary)
Word = RAM(Base + 1, 16 bits)

20

```
BitPos = BitPos + 17
```

```
LastUse(Word) = 0
```

25

RAM(Base) = Word	-- Write First Word to Output Page
LastUse(NextIndex) = Word	-- Write First Word to Dictionary

```
Position = 2
NextIndex = 1
```

30

-- Process Page in units of 2-Byte words

Do Until Position ≥ PageSize

```
{
```

– Processing for Word Not Already in the Dictionary
If RAM(BitPosition, 1 Bit) = "0"

35

```
{
```

```
Word = RAM(BitPosition+1, 16 bits)
```

RAM(Base+Position) = Word	-- Write Word to Output Page
---------------------------	------------------------------

LastUse(NextIndex) = Word	-- Write Word to Dictionary
---------------------------	-----------------------------

40

```
NextIndex = NextIndex + 1
```

45

```
Position = Position + 2
BitPosition = BitPosition + 17
}
```

– Special Handling for Audio Data and the like:
Elseif RAM(BitPosition, 3 Bits) = "110"
{

- 77 -

```

Dif = RAM(BitPosition + 3, 11 bits)
Dif = Dif with 10th bit replicated in bits 12 through 15
Word = RAM(Position - 4) + Dif

5      RAM(Base+Position) = Word    -- Write Word to Output Page
LastUse(NextIndex) = Word    -- Write Word to Dictionary
NextIndex = NextIndex + 1

10     Position = Position + 2
      BitPosition = BitPosition + 14
      }

      - Processing for Single Word Already in the Dictionary
      Elself RAM(BitPosition, 2 Bits) = "10"
15      {
          -- Check for very small Dictionary
          If NextIndex ≤ 2
              { Index = RAM(BitPosition+2, 1 Bit) }
          Else
20          {
              N = IntegerPortion( Log2(NextIndex) ) + 1
              Boundary = 2N - NextIndex
              Index = RAM(BitPosition+2, N-1 Bits)
              If Index ≥ Boundary
                  { Index == RAM(BitPosition+2, N Bits) - Boundary }
              Else
                  { N = N - 1 }
          }

30      Word = LastUse(Index)    -- Get Word from Dictionary
      RAM(Base+Position) = Word    -- Write Word to Output Page

      Position = Position + 2
      BitPosition = BitPosition + 2 + N
35      }

      - Processing for MultiWord Repeat of Data Earlier in Page
      Elself RAM(BitPosition, 3 Bits) = "111"
40      {
          -- Determine Run Length (Number of Contiguous Words Repeated)
          If RAM(BitPosition+3, 1 Bit) = "0"
              {
                  Length = 2
                  BitPosition = BitPosition + 4
              }
          Elself RAM(BitPosition+3, 2 Bits) = "10"
              {
                  Length = 3
              }
      }
  
```

- 78 -

```
      BitPosition = BitPosition + 5
    }
Elseif RAM(BitPosition+3, 3 Bits) = "110"
{
  Length = 4
  BitPosition = BitPosition + 6
}
Elseif RAM(BitPosition+3, 3 Bits) = "111"
{
  Length = 2 + RAM(BitPosition+3, 5 bits)
  BitPosition = BitPosition + 11
}

-- Determine Distance Back of First Repeated Word
If RAM(BitPosition, 2 Bits) = "00"
{
  DistBack = 1 + RAM(BitPosition+2, 4 bits)
  BitPosition = BitPosition + 6
}
Elseif RAM(BitPosition, 2 Bits) = "01"
{
  DistBack = 17 + RAM(BitPosition+2, 6 bits)
  BitPosition = BitPosition + 8
}
Elseif RAM(BitPosition, 1 Bit) = "1"
{
  DistBack = RAM(BitPosition+1, 11 bits)
  BitPosition = BitPosition + 12
}

-- Copy Repeated Data to New Position
Source = Base + Position - (2 x DistBack)
Dest = Base + Position
Copy from RAM(at: Source, through: Source + (2 x Length) - 1)
      to RAM(at: Dest, through: Dest + (2 x Length) - 1)

Position = Position + (Length x 2)

} -- End of Processing for MultiWord Repeat of Earlier Data
} -- End of Main Loop
Return
```

WHAT IS CLAIMED IS:

1. A virtual memory addressing subsystem for a computer having a central processing unit (CPU), and primary and secondary physical memory for storing pages of memory associated with a defined address space having an associated set of virtual memory pages, wherein said defined address space is larger than said primary physical memory's storage capacity; said virtual memory addressing subsystem comprising:
 - 10 a page table for storing a status value for each virtual memory page of said set of virtual memory pages; said page table including a page table entry corresponding to each said virtual memory page, each said page table entry specifying a tag value corresponding to a location in said primary or secondary physical memory; each said page table entry further specifying a plurality of status flags for said corresponding virtual memory page, including a MappedIn flag having a True/False value that indicates when said tag value corresponds to a page in said primary physical memory in which said corresponding virtual memory page is stored;
 - 15 a virtual memory unit (VMU) that translates CPU specified virtual memory address values into primary physical address values when said MappedIn flag in said page table entries corresponding to said CPU specified virtual memory address values are True, and that generates fault signals when said MappedIn flag in said page table entries corresponding to said CPU specified virtual memory address values are False; and
 - 20 a virtual memory manager that responds to each fault signal from said VMU by updating the page table entry corresponding to the CPU specified virtual memory address value which caused said each fault signal so that said page table entry specifies one of said primary physical memory pages and a MappedIn flag having a True value;
 - 25 said virtual memory manager including:
 - 30 a work space manager that defines a work space and a MappedOut storage space for storing said virtual memory pages, said work space comprising

- 80 -

a first portion of said primary physical memory pages and said MappedOut storage space comprising a second portion of said primary physical memory pages and a portion of said secondary physical memory;

- 5 a memory usage monitor that stores memory usage data;
- 10 said work space manager including SwapOut selection logic that selects, on the basis of said memory usage data, ones of said virtual memory pages to be swapped out from said work space to said MappedOut storage space; and
- 15 a MappedOut storage manager that receives from said memory usage manager said virtual memory pages selected to be swapped out, stores said received virtual memory pages in said MappedOut storage space, and adds the primary physical memory pages in which said received virtual memory pages were stored to said MappedOut storage space;
- 20 said MappedOut storage manager including a data compressor that compresses at least some of said received virtual memory pages prior to their storage in said MappedOut storage space, and a data decompressor that decompresses said compressed virtual memory pages when said virtual memory manager responds to fault signals caused by VMU faults on said compressed virtual memory pages;
- 25 said work space manager including work space size logic that determines, on the basis of said memory usage data, when more pages of said primary physical memory are needed in said work space and that dynamically increases said work space by removing pages of said primary physical memory from said MappedOut storage space and adding said pages to said work space.
- 30 2. The virtual memory addressing subsystem of claim 1, wherein said memory usage data stored by said memory usage monitor indicates which of said virtual memory pages in said work space were least recently used.
- 30 3. The virtual memory addressing subsystem of claim 1, wherein

said memory usage data stored by said memory usage monitor indicates which of said virtual memory pages in said work space were least recently used and also indicates how many virtual memory pages were accessed by said CPU during an epoch whose beginning and end are established by predefined criteria;

5 and

10 said work space logic determines when more pages of said primary physical memory are need in said work space by comparing how many virtual memory pages were accessed by said CPU in one said epoch with how many pages are included in said work space.

15

4. The virtual memory addressing subsystem of claim 1, wherein
said MappedOut storage manager stores each said received virtual memory page as compressed data when said data compressor compresses said each received virtual memory page by at least a predefined threshold
15 compression ratio and otherwise stores said each received virtual memory page as uncompressed data.

20

5. The virtual memory addressing subsystem of claim 1, wherein:
said MappedOut storage manager includes logic for preferentially storing
20 said received virtual memory pages in said second portion of said primary physical memory so long as said MappedOut storage page includes a sufficient number of pages of said primary physical memory to store said received virtual memory pages, and for storing ones of said received virtual memory pages in said portion of said secondary memory when said work space size logic in said
25 work space manager determines that more pages of said primary physical memory are needed in said work space.

30

6. A virtual memory addressing method for a computer having a central processing unit (CPU), primary and secondary physical memory for storing pages of memory, and a virtual memory unit, the steps of the method comprising:

- 82 -

- (A) defining an address space having an associated set of virtual memory pages, wherein said defined address space is larger than said primary physical memory's storage capacity;
- 5 (B) defining a work space comprising a first portion of said primary physical memory pages, and allocating each of said primary physical memory pages in said work space to a distinct one of said virtual memory pages; wherein some of said virtual memory pages do not have a primary physical memory page allocated thereto;
- 10 (C) defining a mapped out storage space comprising a second portion of said primary physical memory pages and a portion of said secondary physical memory, wherein said first and second portions of said primary physical memory are mutually exclusive;
- 15 (D) storing in said mapped out storage space those of said virtual memory pages (mapped out virtual memory pages) in which said CPU has stored data and which do not have a primary physical page in said work space allocated thereto;
- 20 (E) dividing said second portion of said primary physical memory into occupied space, in which ones of said mapped out virtual memory pages are stored, and free space, in which no mapped out virtual memory pages are stored; said storing step including (F) compressing at least a subset of said mapped out virtual memory pages using a predefined data compression procedure to produce a compressed virtual memory page, storing said compressed virtual memory pages in said second portion of said primary physical memory if said free space has sufficient storage space to store said compressed virtual memory pages, and otherwise storing said compressed virtual memory pages in said portion of said secondary physical memory; and
- 25 (G) responding to CPU specified virtual memory address values
- (G1) when said CPU specified virtual memory address values are in said pages of said virtual memory to which said ones of said primary physical memory pages have been allocated, by translating said CPU specified virtual memory address values into primary physical address values,

(G2) when said CPU specified virtual memory address values correspond to said mapped out virtual memory pages, allocating ones of said primary physical memory pages in said first or second portions of said primary physical memory pages to each of said corresponding mapped out virtual memory pages, storing said corresponding mapped out virtual memory pages in said allocated ones of said primary physical memory pages and then translating said CPU specified virtual memory address values into primary physical address values in said allocated ones of said primary physical memory pages, and

(G3) when said CPU specified virtual memory address values correspond to virtual memory pages to which no primary physical memory pages have been allocated and which do not correspond to said mapped out virtual memory pages, allocating ones of said primary physical memory pages in said first or second portions of said primary physical memory pages to each of said virtual memory pages in which said CPU specified virtual memory values are located, and then translating said CPU specified virtual memory address values into primary physical address values in said allocated ones of said primary physical memory pages.

7. The method of claim 6,
 - 20 said storing step G2 including, when said CPU specified virtual memory address values correspond to ones of said compressed virtual memory pages, decompressing said corresponding compressed virtual memory pages while performing said storing step in said step G2.
- 25 8. The method of claim 6,
 - 20 said step D including: storing status information for each of said mapped out virtual memory pages, said status information for each of said mapped out virtual memory page indicating where in said mapped out storage space said mapped out virtual memory page is stored and also indicating whether said mapped out virtual memory page is stored as one of said compressed virtual memory pages; and

- 84 -

said step G2 including: inspecting said stored status information to determine whether said CPU specified virtual memory address values correspond to ones of said compressed virtual memory pages, and when said status information so indicates, decompressing said corresponding compressed virtual
5 memory pages while performing said storing step in said step G2.

9. The method of claim 6,

10 said responding step (G) including, in accordance with predefined memory usage criteria, removing ones of said primary physical memory pages from said second portion of said primary physical memory pages and adding those primary physical memory pages to said work space,

whereby said first and second portions of said primary physical memory pages are variable in size.

15 10. The method of claim 6,

further including storing memory usage data indicative of which virtual memory pages have been accessed by said CPU;

20 said responding step (G) including, in accordance with said stored memory usage data and predefined memory usage criteria, removing ones of said primary physical memory pages from said second portion of said primary physical memory pages and adding those primary physical memory pages to said work space,

whereby said first and second portions of said primary physical memory pages are variable in size.

25 11. The method of claim 6,

further including storing memory usage data indicative of which virtual memory pages have been accessed by said CPU, wherein said stored memory usage data indicates which of said virtual memory pages, to which ones of said primary physical memory pages have been allocated, were least recently used;

30 said responding step (G) including, in accordance with said stored memory usage data and predefined memory usage criteria, removing ones of said primary

- 85 -

physical memory pages from said second portion of said primary physical memory pages and adding those primary physical memory pages to said work space, whereby said first and second portions of said primary physical memory pages are variable in size.

5

12. The method of claim 11,

wherein said stored memory usage data also indicates how many virtual memory pages were accessed by said CPU during an epoch whose beginning and end are established by predefined criteria; and

10 said responding step includes determining when more pages of said primary physical memory are need in said work space by comparing how many virtual memory pages were accessed by said CPU in one said epoch with how many pages are included in said work space.

15 13. The method of claim 12, further including

storing data denoting which of at least a predefined subset of said mapped out virtual memory pages were stored first in said second portion of said primary physical memory pages, and

20 said removing step including moving said first stored mapped out virtual memory pages from said second portion of said primary physical memory pages to said portion of said secondary physical memory.

14. The method of claim 6,

further including, responding to a deallocation of a CPU specified virtual memory address range by a program being executed by said CPU by filling said CPU specified virtual memory address range with predefined highly compressible data.

1/11

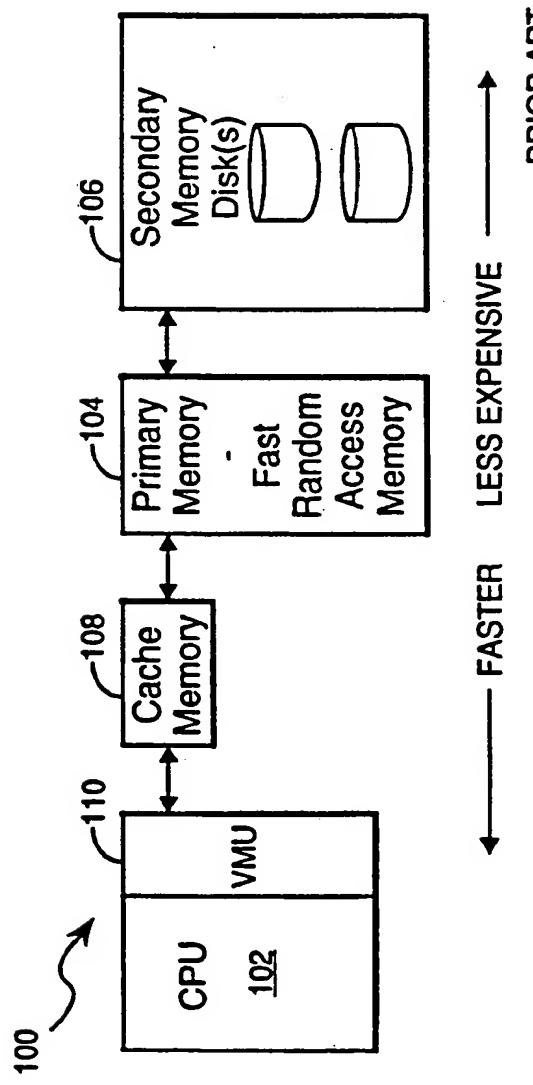


FIGURE 1

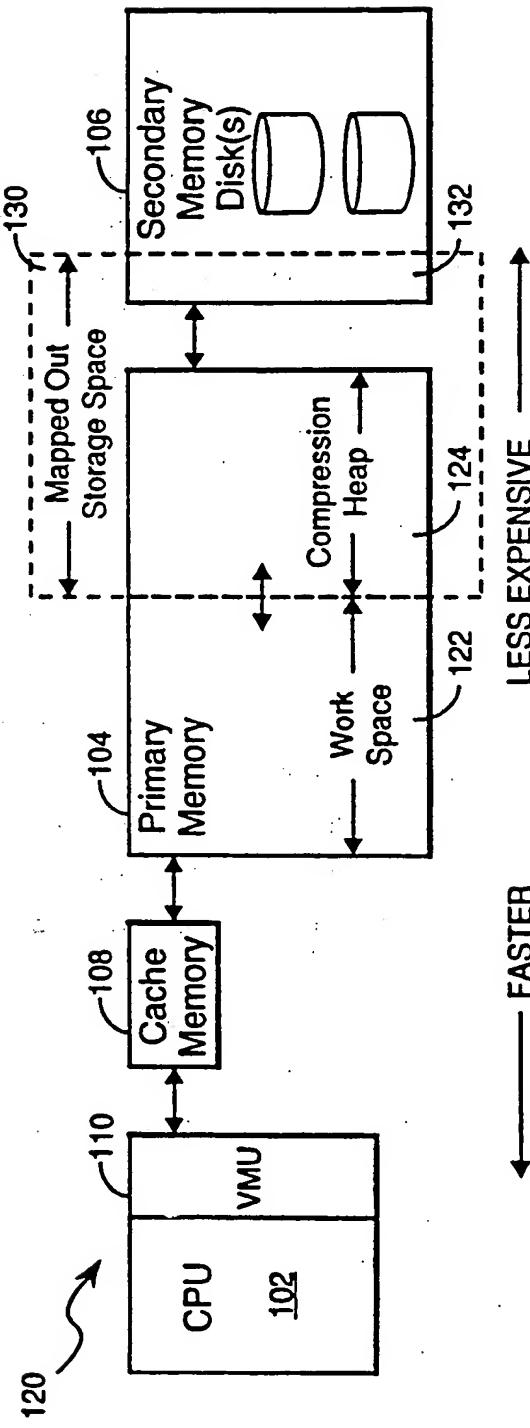


FIGURE 2

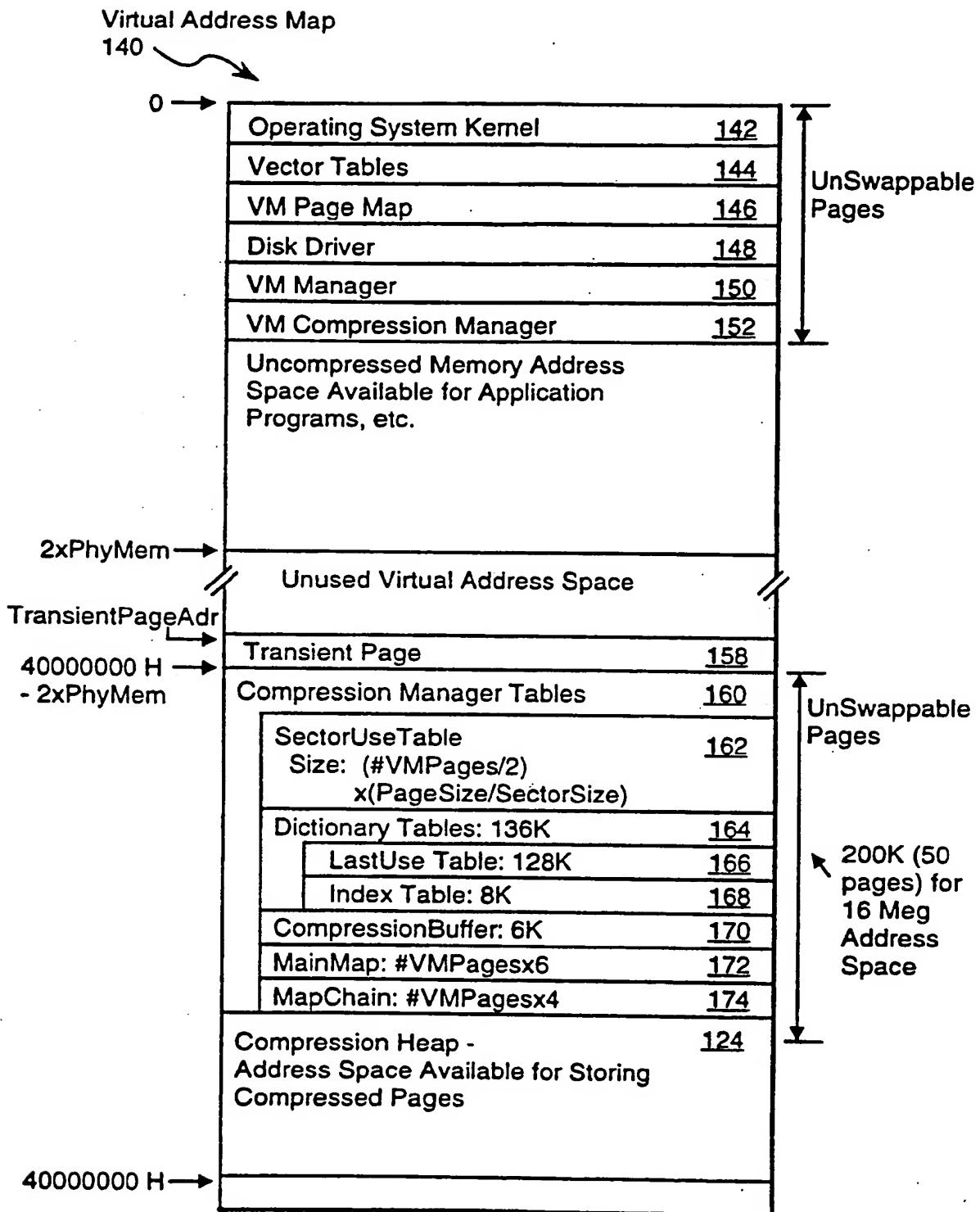


FIGURE 3

3/11

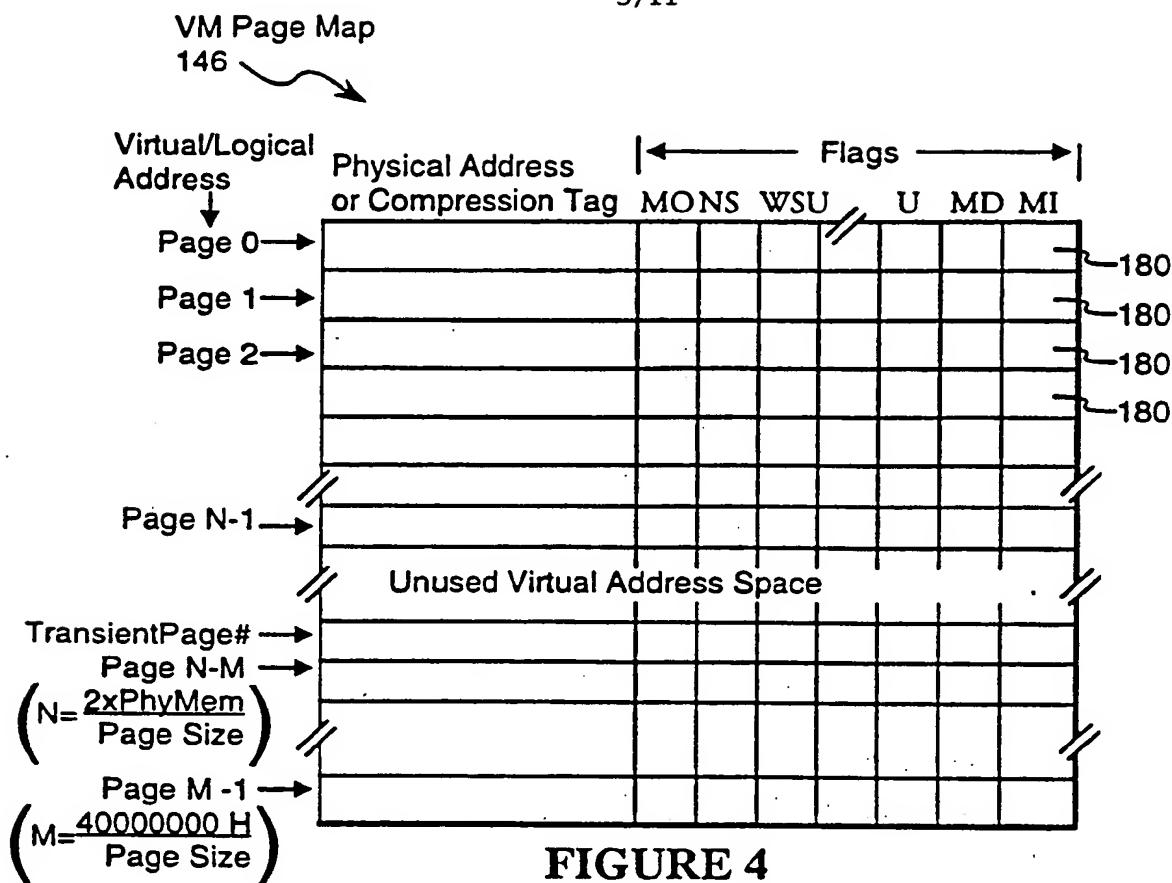
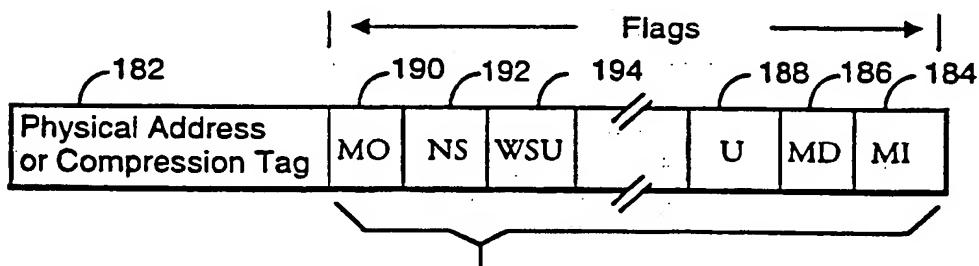


FIGURE 4



- MI** = 1 if Page is Mapped Into Uncompressed Primary Memory
0 otherwise (e.g., Unused, Compressed, etc.)
- MD** = 1 if Page has been modified (written to) since it was last allocated a page in Primary Memory
0 otherwise (e.g., Page has not been modified)
- MO** = 1 if Page is Mapped Out of the Work Space
0 otherwise (e.g., Mapped In or Unused)
- NS** = 1 if Page cannot be Swapped Out of the Work Space
0 otherwise (e.g., Page can be swapped out)
- U** = 1 if Page Used since last time Used Flags were cleared
0 if Page Not Used Since Used Flags were cleared
- WSU** = 1 if Page Used in Current Usage Epoch
0 if Page Not Used in Current Usage Epoch

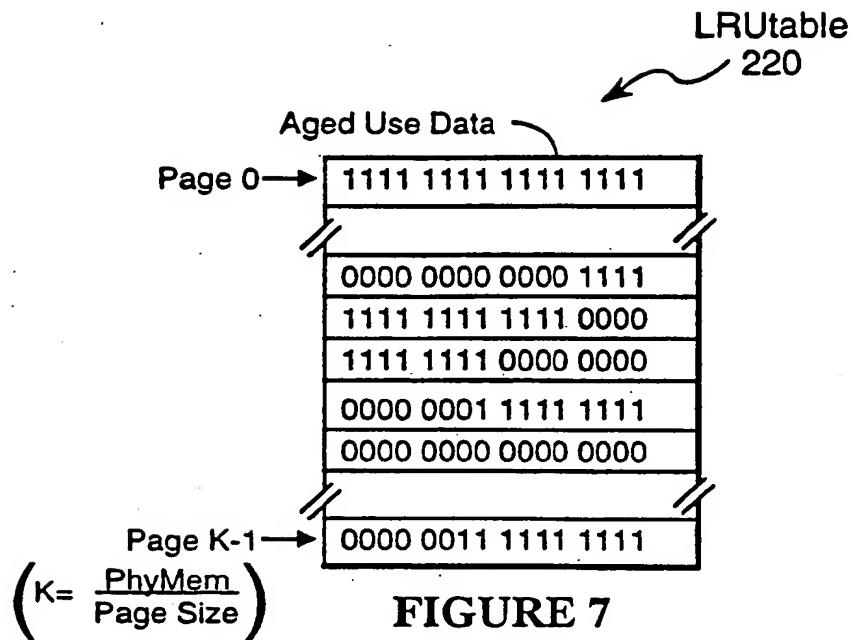
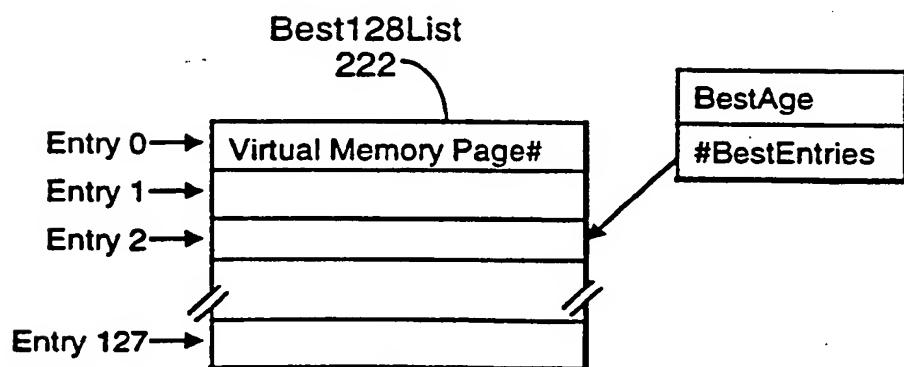
FIGURE 5

4/11

VM Manager	
150	
Executable VM Manager Code	
Initialize - VM Manager Initialization Procedure	216
ReceiveFault - Procedure for responding to VMU Page Fault	200
GetPage() - Procedure for getting RAM page to use	202
MapPage - Procedure for mapping RAM page into fault page	203
SwapOutPage() - Procedure for swapping out a RAM page	204
AgingEvent - LRU Procedure for Selecting Pages to Swap Out	206
BuildBest128List	208
Read Pending Procedure	210
DeallocateSpace Procedures	212
ElapsedEpochTimer - Timer Procedure	214
:	
VM Manager Data	
Best128List	222
LRU - Page Usage Data Table	220
#VMPages (2xPhysical RAM Pages)	
#PagesUsed	
#Uncompressed Pages (excluding Unswappable Pages)	
GrowWS Flag	224
StartTime	
ElapsedEpochTime	
EpochThreshold	
SwapThreshold	
#Swaps	
StartSweepPtr	
XP	
BestAge	
#BestEntries	
Page#	
PG	
LA	
fail	
Flag1	
Flag2	

FIGURE 6

5/11

**FIGURE 7****FIGURE 8**

6/11

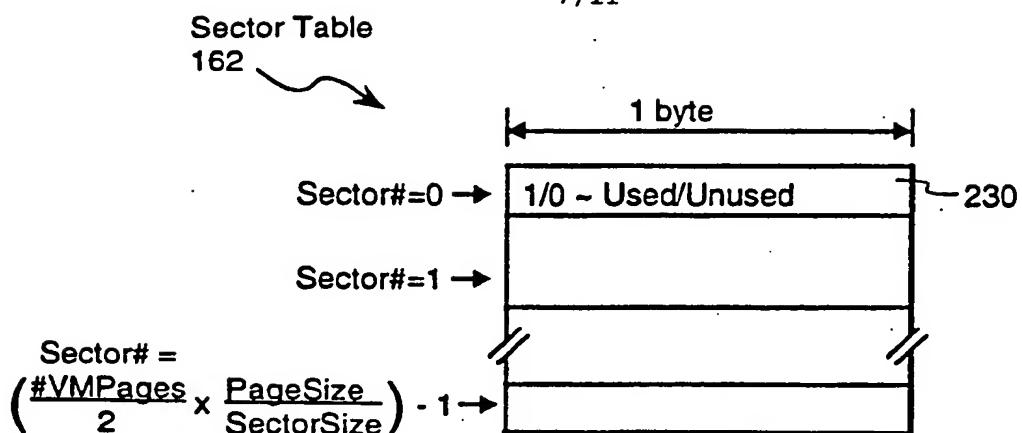
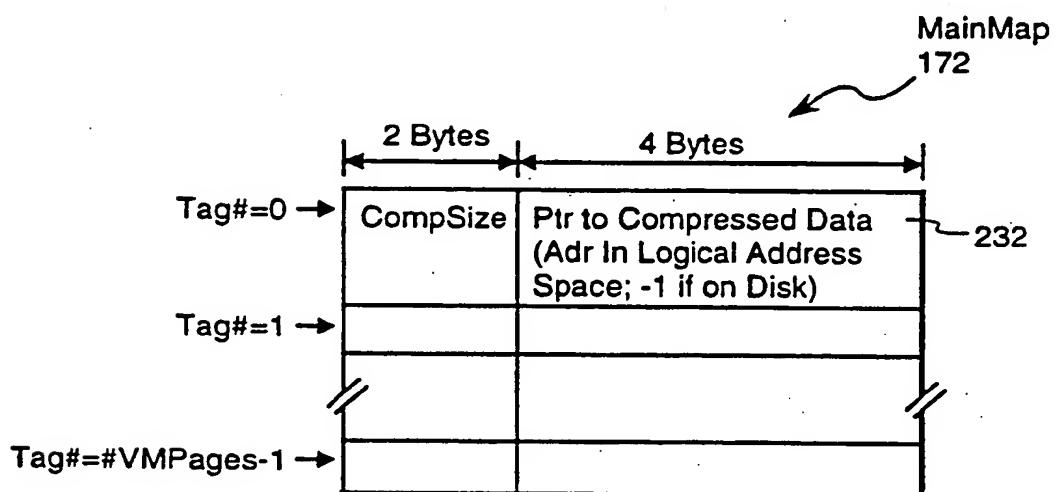
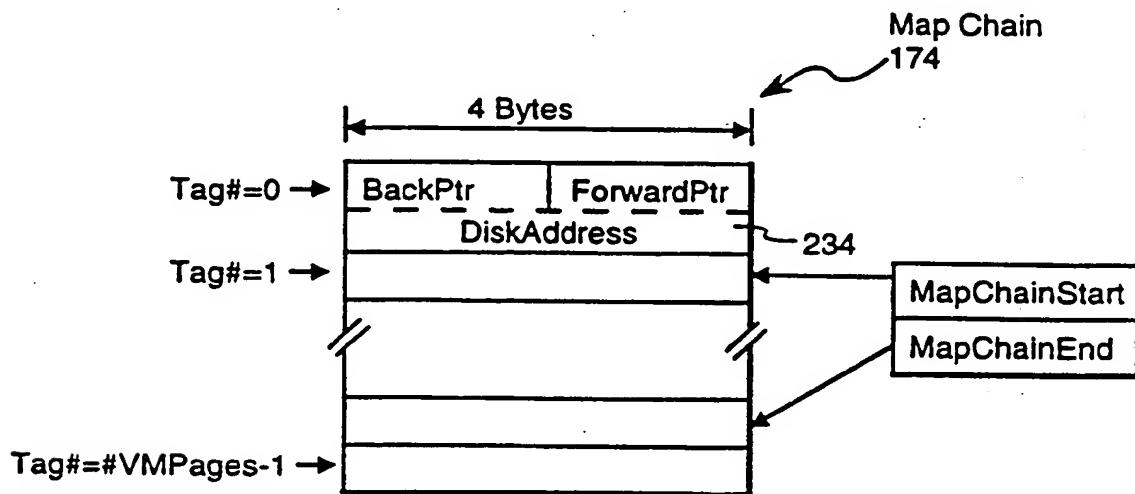
Compression Manager

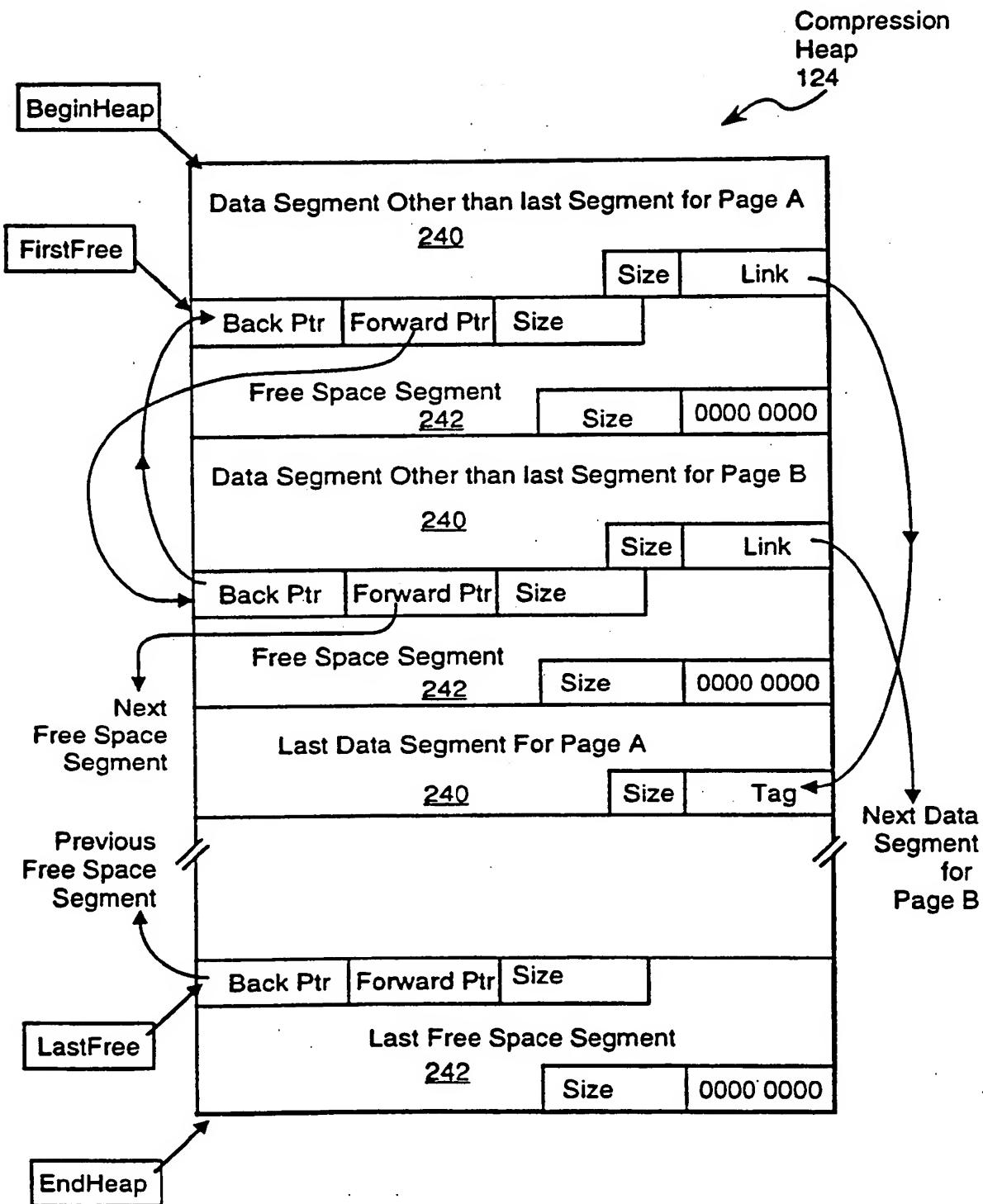
152

Executable Compression Manager Code	
GetCompPage() - Send Free Page to VM Manager	250
FillFromBottom	252
WriteCompPage - Store Page in Compressed Store	254
ReadCompPage - Retreive and Decompress Page from Compressed Store	256
ReleaseCompPage - Release Page in Compressed Store	258
RemoveFromMapChain	260
Compress - Compress one page	262
Decompress - Decompress one page	264
•	
•	
Compression Manager Data	
PtrToDiskMap - Ptr to sector table	
PtrFreeSector	
PtrMapChain	
PtrCompWorkSpace	
PtrCompBuffer	
PtrMainMap	
FreeMapEntry	
PtrEndMap	
FirstFree - Ptr to First Free Space Block in Heap	
LastFree - Ptr to Last Free Space Block in Heap	
EndHeap - Ptr to end of Heap	
FreeHeapBytes - number of free bytes in the Heap	
NextCompHeapPage - logical page # for next page to be added to heap	
MapChainStart - beginning of MapChain	
MapChainEnd - end of Map Chain	
GrowWS	
SwapOutFlag	
Word	
Flag3	
Flag4	
Page#	
PG	
fail	

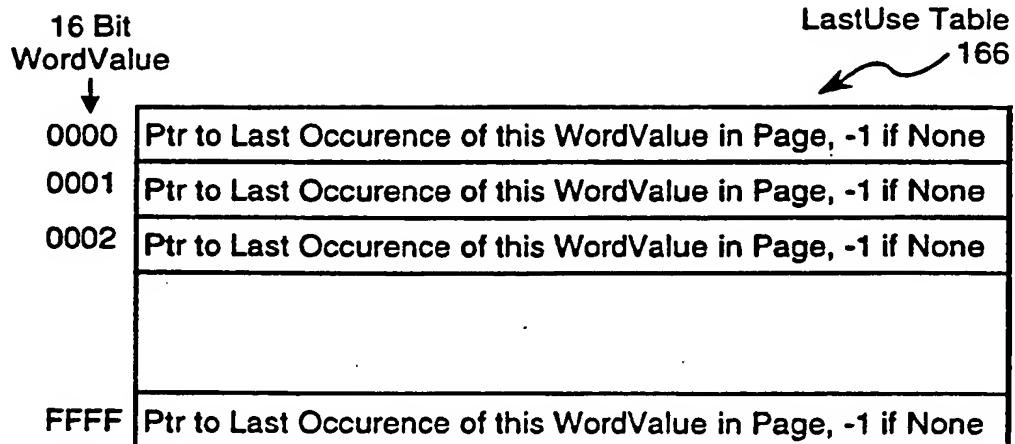
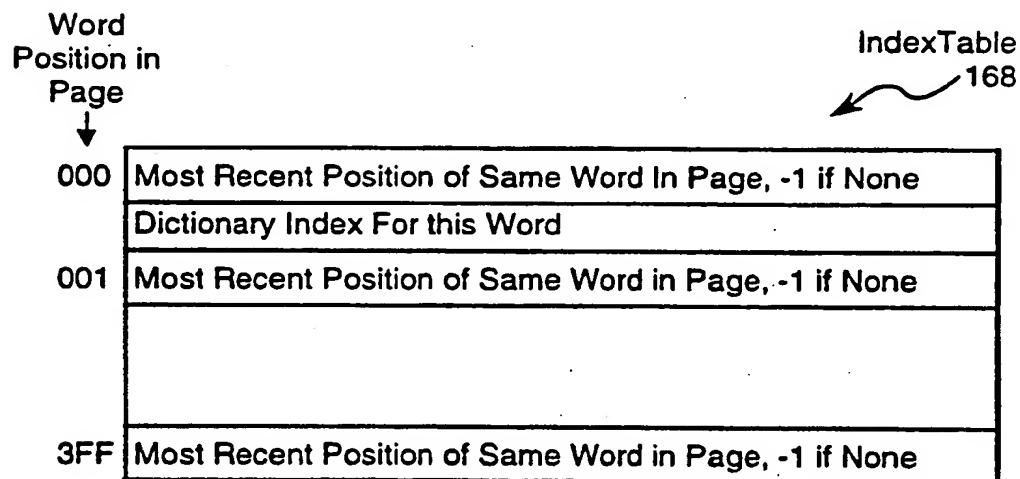
FIGURE 9

7/11

**FIGURE 10****FIGURE 11****FIGURE 12**

**FIGURE 13**

9/11

**FIGURE 14****FIGURE 15**

10/11

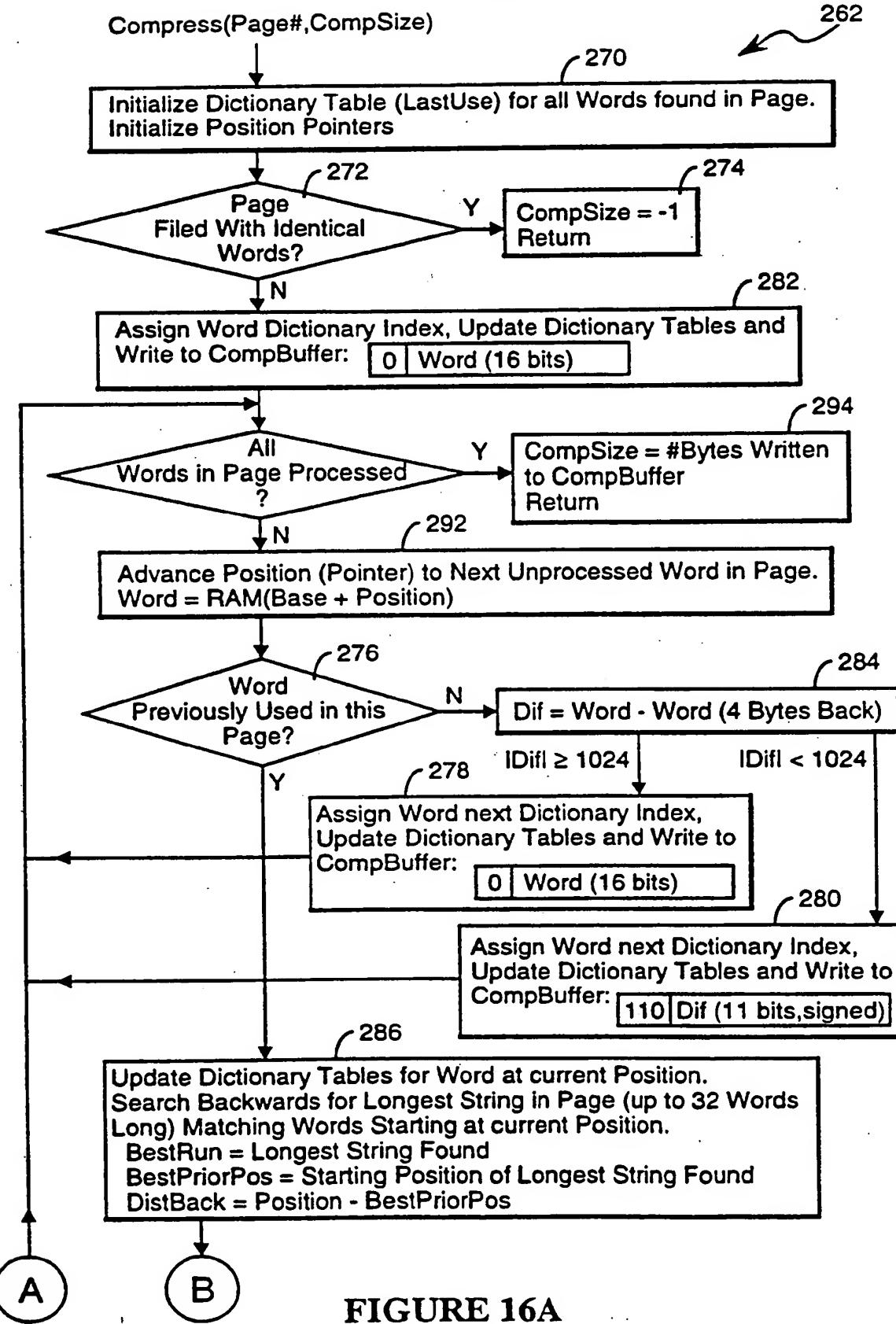


FIGURE 16A

11/11

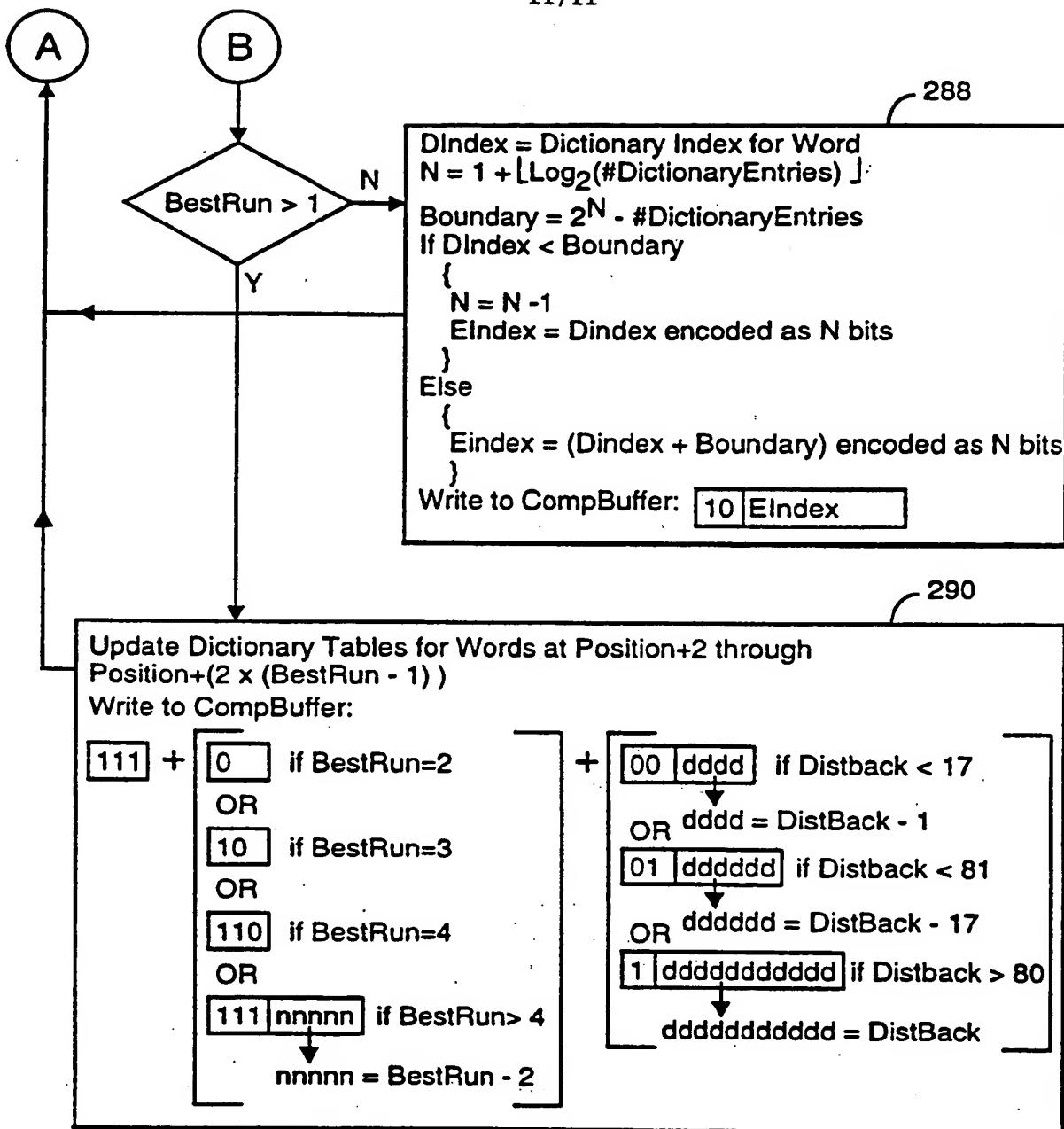


FIGURE 16B

FIGURE 16A

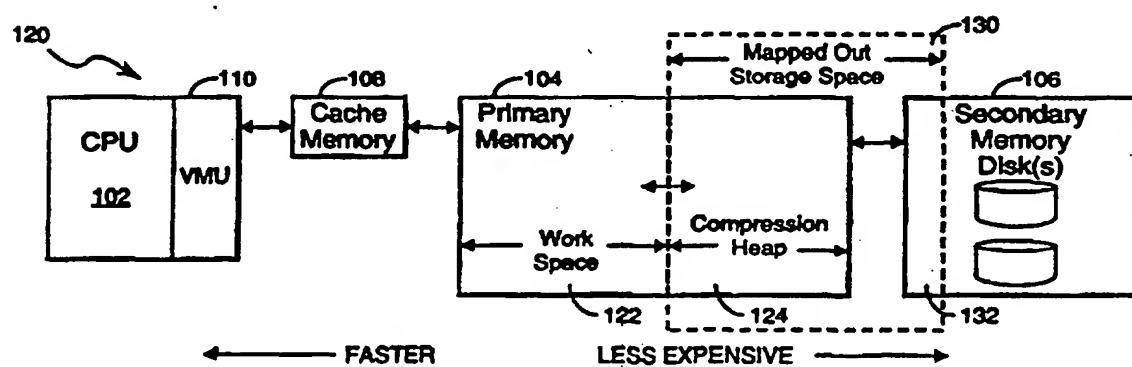
FIGURE 16B

FIGURE 16



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ :	A3	(11) International Publication Number:	WO 95/18997
G06F 12/08, 13/14		(43) International Publication Date:	13 July 1995 (13.07.95)
(21) International Application Number:	PCT/US94/14987	(81) Designated States:	AM, AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, JP, KE, KG, KP, KR, KZ, LK, LR, LT, LU, LV, MD, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SI, SK, TJ, TT, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG), ARIPO patent (KE, MW, SD, SZ).
(22) International Filing Date:	29 December 1994 (29.12.94)		
(30) Priority Data:	08/177,622 30 December 1993 (30.12.93) US		
(71) Applicant:	CONNECTIX CORPORATION [US/US]; 2655 Campus Drive, San Mateo, CA 94403 (US).		
(72) Inventors:	GARBER, Jonathan, F.; 13503 Skyline Boulevard, Woodside, CA 94062 (US). BROWN, Jorg, A.; 1516 Wicket Court, Concord, CA 94518 (US). WALTERS, Chad, Perry; 550 Middlefield Avenue, Palo Alto, CA 94301 (US).		
(74) Agents:	WILLIAMS, Gary, S. et al.; Flehr, Hohbach, Test, Albritton & Herbert, 4 Embarcadero Center, Suite 3400, San Francisco, CA 94111-4187 (US).		
(54) Title:	VIRTUAL MEMORY MANAGEMENT SYSTEM AND METHOD USING DATA COMPRESSION		



(57) Abstract

A virtual memory system and method enable a computer system (120) to use a virtual memory address space larger than the size of physical primary memory (104) while swapping few, if any, pages out to secondary memory (106). Primary memory (104) is divided into a work space (122), used for storing uncompressed pages in current use, and a "Compression Heap" (124). A MappedOut storage space (130), which includes the Compression Heap and a portion of secondary memory, is used to store all pages swapped out of the work space (122). A virtual memory manager (110) dynamically determines the number of pages of primary memory (104) which need to be included in the work space, and moves pages of primary memory into the work space from the Compression Heap (124) as needed. Pages are selected to be swapped out of the work space to the MappedOut storage space (130), and a data decompressor decompresses the compressed virtual memory pages when a fault occurs on those pages. The virtual memory manager (110) includes logic for preferentially compressing and storing swapped out virtual memory pages in the Compression Heap (124) so long as the MappedOut storage space (130) includes a sufficient number of pages of primary memory to store those virtual memory pages, and for storing ones of the swapped out virtual memory pages in secondary memory (106) when insufficient space is available in the Compression Heap (124).

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	GB	United Kingdom	MR	Mauritania
AU	Australia	GE	Georgia	MW	Malawi
BB	Barbados	GN	Guinea	NE	Niger
BE	Belgium	GR	Greece	NL	Netherlands
BF	Burkina Faso	HU	Hungary	NO	Norway
BG	Bulgaria	IE	Ireland	NZ	New Zealand
BJ	Benin	IT	Italy	PL	Poland
BR	Brazil	JP	Japan	PT	Portugal
BY	Belarus	KE	Kenya	RO	Romania
CA	Canada	KG	Kyrgyzstan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo	KR	Republic of Korea	SE	Sweden
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LJ	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LU	Luxembourg	TD	Chad
CS	Czechoslovakia	LV	Latvia	TG	Togo
CZ	Czech Republic	MC	Monaco	TJ	Tajikistan
DE	Germany	MD	Republic of Moldova	TT	Trinidad and Tobago
DK	Denmark	MG	Madagascar	UA	Ukraine
ES	Spain	ML	Mali	US	United States of America
FI	Finland	MN	Mongolia	UZ	Uzbekistan
FR	France			VN	Viet Nam
GA	Gabon				

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US94/14987

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G06F 12/08, 13/14
 US CL : 395/400, 425

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/400, 425

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US, A, 5,237,460 (Miller et al.) 29 JANUARY 1991, entire document	1-14
A	US, A, 4,989,134 (Shaw et al.) 29 JANUARY 1991, entire document	1-14
A	US, A, 5,101,485, (Perrazoli, Jr.) 31 MARCH 1992, entire document	1-14
A	IEEE COMPUTER, VOLUME 15, NO. 3, H. M. LEVY ET AL., MARCH 1982, "VIRTUAL MEMORY MANAGEMENT IN THE VAX/VMS OPERATING SYSTEM", PAGES 35-41	1-14

 Further documents are listed in the continuation of Box C. See patent family annex.

* Special categories of cited documents:	T	later documents published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
A document defining the general state of the art which is not considered to be part of particular relevance	X	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
E earlier document published on or after the international filing date	Y	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
L document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	&	document member of the same patent family
O document referring to an oral disclosure, use, exhibition or other means		
P document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search

04 April 1995

Date of mailing of the international search report

31 JUL 1995

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. NOT APPLICABLE

Authorized officer

MICHAEL A. WHITFIELD

Telephone No. (703) 305-9600

Form PCT/ISA/210 (second sheet)(July 1992)*

